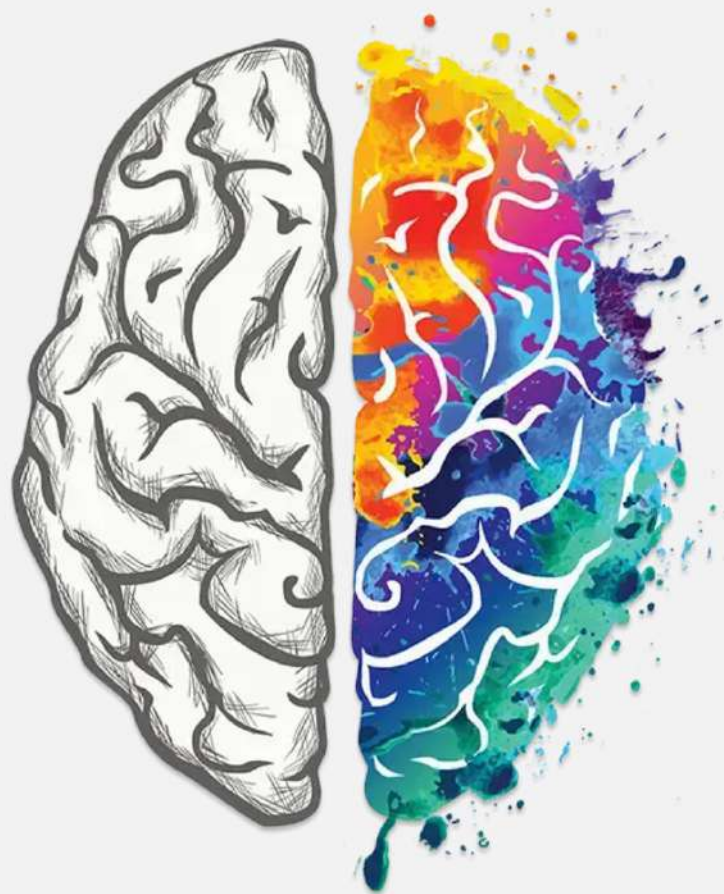


# DSA

## Beginners's BRAIN



*a complete data structure  
and algorithms guide  
for competitive  
programmers.*

# Stacks & Queues

- Karun Karthik

## Contents

0. Introduction
1. Implementation of stack using Linkedlist
2. Implementation of queue using Linkedlist
3. Implementation of stack using queue
4. Implementation of queue using stack
5. Valid Parenthesis
6. Asteroid Collision
7. Next Greater Element
8. Next Smaller Element
9. Stock Span Problem
10. Celebrity Problem
11. Largest Rectangle in Histogram
12. Sliding Window Maximum

## Stack → Linear data structure

- follows LIFO, last in first out.
- operations → push: insert into top of stack  
pop: delete from top of stack.

### Applications →

- by compilers to check for parenthesis
- to evaluate postfix expression
- to convert infix to postfix / prefix form.
- to store values during recursion & context during function call.
- to implement DFS of graph

## Queue → Linear data structure

- follows FIFO, first in first out.
- operations → enqueue: insert element at end of queue  
dequeue: delete element at start of queue

### Applications →

- schedule jobs by CPU.
- to carry out FIFO basis like printing jobs.
- to implement BFS of graph

### Types →

- Queue
- Circular Queue
- Doubly ended Queue
- Priority Queue.

# ① Implement a Stack using Linkedlist →

code →

```
1  #include <bits/stdc++.h>
2  using namespace std;
3
4  struct Node{
5      int data;
6      Node* next;
7  };
8
9  Node* top;
10
11 void push(int data){
12     Node* temp = new Node();
13     if (!temp){
14         cout << "\nStack Overflow";
15         exit(1);
16     }
17     // add at the top and change top as new node
18     temp->data = data;
19     temp->next = top;
20     top = temp;
21 }
22
23 int isEmpty(){
24     // if top is null then empty
25     return top == NULL;
26 }
27
28 int peek(){
29     // if stack is not empty then return top node's data
30     if (!isEmpty())
31         return top->data;
32     else
33         exit(1);
34 }
35
36 void pop(){
37     Node* temp;
38     if (top == NULL){
39         cout << "\nStack Underflow" << endl;
40         exit(1);
41     } else {
42         temp = top;
43         top = top->next;
44         free(temp);
45     }
46 }
47
```



## ② Implement a Queue using LinkedList →

code →

```
1  class Node {
2      int data;
3      Node* next;
4      Node(int d){
5          data = d;
6          next = NULL;
7      }
8  };
9
10 class Queue {
11     Node *front, *rear;
12
13     Queue(){
14         front = rear = NULL;
15     }
16
17     void enqueue(int x)
18     {
19         Node* temp = new Node(x);
20         // if empty then node is both front and rear
21         if (rear == NULL) {
22             front = rear = temp;
23             return;
24         }
25         // else add at end
26         rear->next = temp;
27         rear = temp;
28     }
29
30     void dequeue()
31     {
32         // if empty then return NULL
33         if (front == NULL)
34             return;
35         // store front node
36         Node* temp = front;
37         front = front->next;
38
39         // if front is NULL => no Nodes, change rear to NULL
40         if (front == NULL)
41             rear = NULL;
42         // free node
43         delete (temp);
44     }
45 };
```

### ③ Implement a Stack using Queue →

If push, push into queue from rear end & pop & push all elements  
If pop, pop from queue from front end.

Code →

```
1  class Stack {
2      queue <int> q;
3
4      public:
5
6          // push operation
7          void Push(int x) {
8              int n = q.size();
9              q.push(x);
10             for (int i = 0; i < n; i++)
11             {
12                 int value = q.front();
13                 q.pop();
14                 q.push(value);
15             }
16         }
17
18         // pop operation
19         int Pop() {
20             int value = q.front();
21             q.pop();
22             return value;
23         }
24
25         // accessing top value
26         int Top() {
27             return q.front();
28         }
29
30         // finding size of stack
31         int Size() {
32             return q.size();
33         }
34     };
35
```

#### ④ Implement a Queue using Stack →

→ use 2 stacks.

→ while pop(), shift all elements in 1 stack to another.  
& return top value.

code →

```
1  class Queue {
2      public:
3          stack<int> in;
4          stack<int> out;
5
6          // push operation
7          void Push(int x) {
8              in.push(x);
9          }
10
11         // pop operation
12         int Pop() {
13             // shift in to out
14             if (out.empty()){
15                 while (in.size()){
16                     out.push(in.top());
17                     in.pop();
18                 }
19             }
20             int x = out.top();
21             out.pop();
22             return x;
23         }
24
25         // peek operation
26         int Top() {
27             if (out.empty()){
28                 while (in.size()){
29                     out.push(in.top());
30                     in.pop();
31                 }
32             }
33             return out.top();
34         }
35
36         int Size() {
37             return in.size()+out.size();
38         }
39     };
```

# ⑤ Valid parenthesis

$$S = \{ \} \rightarrow T$$

$$S = \{ [ ] \} \rightarrow T$$

$$S = \{ ( ) \{ \} \} \rightarrow T$$

$$S = \{ ) [ \} \rightarrow F$$

Ex  $s = \{ [ ] ( ) \{ ( ) [ ] ( [ ] ) \} \rightarrow \text{True}$ .

→ if match found then pop, else push.

stack : [ ]	s = { [ ] ( ) { ( ) [ ] ( [ ] ) }
stack : [ {	s = { [ ] ( ) { ( ) [ ] ( [ ] ) }
stack : [ { [	s = { [ ] ( ) { ( ) [ ] ( [ ] ) }
stack : [ { [ ]	s = { [ ] ( ) { ( ) [ ] ( [ ] ) }
stack : [ { (	s = { [ ] ( ) { ( ) [ ] ( [ ] ) }
stack : [ { [ ]	s = { [ ] ( ) { ( ) [ ] ( [ ] ) }
stack : [ { (	s = { [ ] ( ) { ( ) [ ] ( [ ] ) }
stack : [ (	s = { [ ] ( ) { ( ) [ ] ( [ ] ) }
stack : [ ( ]	s = { [ ] ( ) { ( ) [ ] ( [ ] ) }
stack : [ ( [	s = { [ ] ( ) { ( ) [ ] ( [ ] ) }
stack : [ ( [ ]	s = { [ ] ( ) { ( ) [ ] ( [ ] ) }
stack : [ ( [ ]	s = { [ ] ( ) { ( ) [ ] ( [ ] ) }
stack : [ ( [ ]	s = { [ ] ( ) { ( ) [ ] ( [ ] ) }

∴ As the stack is empty & string is completely traversed  
the string is valid ∴ return true.

Code →

```
1  class Solution {
2  public:
3      bool isValid(string s) {
4          stack<char> st;
5          for(auto i : s)
6          {
7              if (st.empty() || i == '(' || i == '{' || i == '[')
8              {
9                  st.push(i);
10             }
11             else
12             {
13                 if ((i == ')' && st.top() != '(') ||
14                     (i == ']' && st.top() != '[') ||
15                     (i == '}' && st.top() != '{')){
16                     return false;
17                 }
18                 st.pop();
19             }
20         }
21         return st.empty();
22     }
23 };
```

$Tc \rightarrow O(n)$

$Sc \rightarrow O(n)$



## ⑥ Asteroid Collision →

only consider magnitude

+ve sign  $\Rightarrow$  right direction

-ve sign  $\Rightarrow$  left direction

if  $x \neq y$  collide then  $\min(x, y)$  will be removed

if  $x = y$  then both will be removed.

Eg  $[5, 10, -5]$  5, 10 will not collide

10, -5 will collide & -5 will be removed

result =  $[5, 10]$

Eg  $[10, 6, -8, -8, 8, 9]$

stack   $[10, 6, -8, -8, 8, 9]$

stack 10  $[10, 6, -8, -8, 8, 9]$

stack 10, 6  $[10, 6, -8, -8, 8, 9]$  as 6 is +ve push

stack 10, 6  $[10, 6, -8, -8, 8, 9]$  as 6 & 8 will collide (opp directions), 6 will be removed

stack 10  $[10, 6, -8, -8, 8, 9]$  as 10 & 8 will collide (opp directions), 8 will be removed

stack 10  $[10, 6, -8, -8, 8, 9]$  as 10 & 8 will collide (opp directions), 8 will be removed

stack 10, 8  $[10, 6, -8, -8, 8, 9]$  as 8 is +ve push

stack 10, 8, 9  $[10, 6, -8, -8, 8, 9]$  as 9 is +ve push

result =  $[10, 8, 9]$

TC  $\rightarrow O(2n) \simeq O(n)$  SC  $\rightarrow O(n)$   
worst case

code →

```
1  class Solution {
2  public:
3      vector<int> asteroidCollision(vector<int>& asteroids) {
4
5          vector<int> res;
6
7          for(int i=0; i< asteroids.size(); i++){
8
9              if(res.empty() || asteroids[i]>0)
10                 res.push_back(asteroids[i]);
11             else {
12
13                 while(!res.empty() && res.back()>0 && res.back()<abs(asteroids[i])) {
14                     res.pop_back();
15                 }
16
17                 if(!res.empty() && res.back()+asteroids[i]==0)
18                     res.pop_back();
19                 else if(res.empty() || res.back()<0)
20                     res.push_back(asteroids[i]);
21             }
22         }
23         return res;
24     }
25 };
```

⑦ Next greater element  $\rightarrow$   $[2, 4, 1, 3, 1, 6]$

Eg  $[4, 5, 2, 25]$

$4 \rightarrow 5$      $2 \rightarrow 25$   
 $5 \rightarrow 25$      $25 \rightarrow -1$

$2 \rightarrow 4$      $3 \rightarrow 6$   
 $4 \rightarrow 6$      $1 \rightarrow 6$   
 $1 \rightarrow 3$      $6 \rightarrow -1$

- $\rightarrow$  Iterate from last & compare its value with top of stack
- $\rightarrow$  If stack is greater than its the next greater element
- $\rightarrow$  else keep popping till the next greater element is found.

Eg  $[11, 13, 3, 10, 7, 21, 26]$

$\leftarrow$

Stack = [	]	$[11, 13, 3, 10, 7, 21, 26]$	
Stack = [26	]	$[11, 13, 3, 10, 7, 21, 26]$	$26 \rightarrow -1$
Stack = [26, 21	]	$[11, 13, 3, 10, 7, 21, 26]$	$21 \rightarrow 26$
Stack = [26, 21, 7	]	$[11, 13, 3, 10, 7, 21, 26]$	$7 \rightarrow 21$
Stack = [26, 21, <del>7</del> , 10	]	$[11, 13, 3, 10, 7, 21, 26]$	pop 7, push 10 $10 \rightarrow 21$
Stack = [26, 21, 10	]	$[11, 13, 3, 10, 7, 21, 26]$	$3 \rightarrow 10$
Stack = [26, 21, <del>10</del> , <del>3</del> , 13	]	$[11, 13, 3, 10, 7, 21, 26]$	pop 3, 10 push 13 $13 \rightarrow 21$
Stack = [26, 21, 13	]	$[11, 13, 3, 10, 7, 21, 26]$	$11 \rightarrow 13$

Ans =  $[13, 21, 10, 21, 21, 26, -1]$

Code →

```
1  class Solution
2  {
3      public:
4          //Function to find the next greater element for each element of the array.
5          vector<long long> nextLargerElement(vector<long long> arr, int n){
6
7              stack<long long> st;
8              vector<long long> res(n);
9
10             for(int i=n-1; i>=0 ; i--){
11                 long long currVal = arr[i];
12
13                 while(!st.empty() && st.top()<=currVal)
14                     st.pop();
15
16                 res[i] = st.empty()? -1:st.top();
17                 st.push(currVal);
18             }
19             return res;
20         }
21     };
22
```

$T_c \rightarrow O(n)$

$S_c \rightarrow O(n)$

## ⑧ Next Smaller element →

→ entire approach is similar to next greater element except for comparison.

code →

$T_c \rightarrow O(n)$

$S_c \rightarrow O(n)$

```
1  vector<int> nextSmallerElement(vector<int> &arr, int n)
2  {
3      stack<int> st;
4      vector<int> res(n);
5      for(int i=n-1; i>=0 ; i--){
6
7          long long currVal = arr[i];
8
9          while(!st.empty() && st.top()>=currVal)
10             st.pop();
11
12             res[i] = st.empty()? -1 : st.top();
13             st.push(currVal);
14     }
15     return res;
16 }
```



⑨ Stock Span Problem → given price quotes of stock for  $n$  days.  
 we need to find span of stock on any particular day.  
 max no. of consecutive days for which price  $\leq$  curr day's price

Eg [100, 80, 60, 70, 60, 75, 85]

stack = [stores indexes]

span = [0 0 0 0 0 0 0]  
 0 1 2 3 4 5 6

if currentElement > stack.top

pop stack

else:

span = currentIndex - stack.top

→ push index into stack after processing →

0 1 2 3 4 5 6

[100, 80, 60, 70, 60, 75, 85] span of 1<sup>st</sup> element = 1

stack

[0]

span

[1 0 0 0 0 0 0]

[100, 80, 60, 70, 60, 75, 85] 80 > 100 ⇒ false  
 ∴ span = 1 - 0 = 1

[0, 1]

[1 1 0 0 0 0 0]

[100, 80, 60, 70, 60, 75, 85] 60 > 100 ⇒ false  
 ∴ span = 2 - 1 = 1

[0, 1, 2]

[1 1 1 0 0 0 0]

[100, 80, 60, 70, 60, 75, 85] 70 > 60 ⇒ true ∴ pop  
 70 > 80 ⇒ false  
 ∴ span = 3 - 1 = 2

[0, 1, 3]

[1 1 1 2 0 0 0]

[100, 80, 60, 70, 60, 75, 85] 60 > 70 ⇒ false  
 ∴ span = 4 - 3 = 1

[0, 1, 3, 4]

[1 1 1 2 1 0 0]

[100, 80, 60, 70, 60, 75, 85] 75 > 60 ⇒ true ∴ pop  
 75 > 70 ⇒ true ∴ pop  
 75 > 80 ⇒ false  
 span = 5 - 1 = 4

[0, 1, 5]

[1 1 1 2 1 4 0]

[100, 80, 60, 70, 60, 75, 85] 85 > 75 ⇒ true ∴ pop  
 85 > 80 ⇒ true ∴ pop  
 85 > 100 ⇒ false  
 span = 6 - 0 = 6

[0, 6]

[1 1 1 2 1 4 6]

span = [1 1 1 2 1 4 6]  
 0 1 2 3 4 5 6

Code →

$T_c \rightarrow O(n)$

$Sc \rightarrow O(n)$

```
1  class Solution
2  {
3      public:
4          //Function to calculate the span of stocks price for all n days.
5          vector<int> calculateSpan(int price[], int n)
6          {
7              vector<int> span(n);
8              stack<int> st;
9
10             st.push(0);
11             span[0] = 1;
12
13             for(int i=1; i<n; i++){
14
15                 int currPrice = price[i];
16
17                 while(!st.empty() && currPrice >= price[st.top()])
18                     st.pop();
19
20                 if(st.empty()){
21                     span[i] = i+1;
22                 } else {
23                     span[i] = i-st.top();
24                 }
25
26                 st.push(i);
27             }
28             return span;
29         }
30     };
31
```

# ⑩ Celebrity Problem →

A Celebrity is a person, who is known to everyone & knows none.

Given a square matrix  $M$  & if  $i^{\text{th}}$  person knows  $j^{\text{th}}$  person then  $M[i][j] = 1$ , else 0.

Eg →  $n = 3$ .

$M = \begin{matrix} & \begin{matrix} 0 & 1 & 2 \end{matrix} \\ \begin{matrix} 0 \\ 1 \\ 2 \end{matrix} & \begin{bmatrix} 0 & 1 & 0 \\ 0 & 0 & 0 \\ 0 & 1 & 0 \end{bmatrix} \end{matrix}$

Stack →  $[\ ] \Rightarrow [0, 1, 2]$

① create stack & push values from 0 to  $n-1$ .  
② do the following till stack more than has 1 value.

- pop 1st element & set it to A
- pop again & set it to B
- if A knows B then push B else A.

⇒  $\overset{\text{stack}}{[0, \cancel{1}, \cancel{2}]} \quad A = 2 \quad \& \quad M[2][\overset{\text{true}}{2}] == 1 \quad \therefore \text{push } 1 \Rightarrow \overset{\text{stack}}{[0, 1]}$   
 $B = 1$

$\overset{\text{stack}}{[\cancel{0}, \cancel{1}]} \quad A = 1 \quad \& \quad M[\overset{\text{false}}{1}][0] == 1 \quad \therefore \text{push } 1 \Rightarrow \overset{\text{stack}}{[1]}$   
 $B = 0$

$\therefore$  as stack has only 1 element, stop.

⇒ Now pop the stack & consider it as celebrity & check for

- anyone doesn't know celeb ( $\neg M[i][\text{celeb}]$ )
- if celeb knows anyone ( $M[\text{celeb}][i]$ )

} return -1.

$\therefore$  from  $i = 0$  to 2 & celeb = 1

$i = 0 \quad (\neg M[0][1] \text{ or } M[1][0]) = 0$   
 $i = 1 \quad \text{skip as celeb is } i$   
 $i = 2 \quad (\neg M[2][1] \text{ or } M[1][2]) = 0$

all are failed i.e. no violation of conditions.

$\therefore$  return celeb i.e. 1

code →

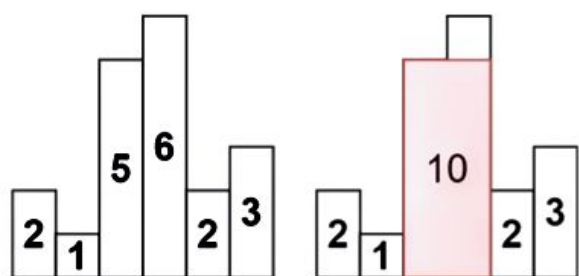
$TC = O(n)$

$SC = O(n)$

```
1  class Solution
2  {
3      public:
4          //Function to find if there is a celebrity in the party or not.
5          int celebrity(vector<vector<int> >& M, int n) {
6
7              stack<int> s;
8
9              for(int i=0;i<n;i++)    s.push(i);
10
11             // check and if is a celebrity then push into stack
12             while(s.size()>1)
13             {
14                 int a=s.top();
15                 s.pop();
16                 int b=s.top();
17                 s.pop();
18
19                 if(M[a][b]==1)
20                     s.push(b);
21                 else
22                     s.push(a);
23             }
24
25             int celeb = s.top();
26
27             for (int i = 0; i < n; i++){
28                 // if i person doesn't know celeb or celeb knows anyone else
29                 // then return -1
30                 if ( (i!=celeb) && (!M[i][celeb]) || M[celeb][i] ))
31                     return -1;
32             }
33
34             return celeb;
35         }
36     };
```



# ⑪ Largest Rectangle in Histogram →



⇒ given an array of heights, return area of largest rectangle

Ans = 10.

0 1 2 3 4 5      Stack.

arr = [2, 1, 5, 6, 2, 3]      [      ]      area = 0      maxArea = 0

i = 0      [2, 1, 5, 6, 2, 3]      [0]      area = 0      maxArea = 0

→ i = 1      [2, 1, 5, 6, 2, 3]      [0]      area = 0      maxArea = 0

now  $arr[st.top()] > curElement \Rightarrow ht = arr[st.top()] \ \& \ st.pop() \uparrow$   
 & as stack is empty now, width = i & push(i)!

∴ ht = 2 & width = 1 ∴ area = 2 & maxArea = 2.

→ i = 2      [2, 1, 5, 6, 2, 3]      [1]      area = 0      maxArea = 2

now  $arr[st.top()] > curElement \Rightarrow false \therefore push(i)^2$

→ i = 3      [2, 1, 5, 6, 2, 3]      [1, 2]      area = 0      maxArea = 2

now  $arr[st.top()] > curElement \Rightarrow false \therefore push(i)^3$

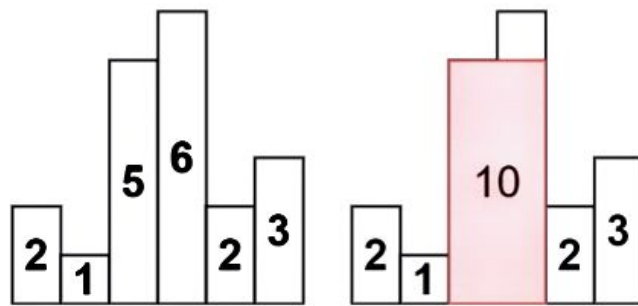
→ i = 4      [2, 1, 5, 6, 2, 3]      [1, 2, 3]      area = 0      maxArea = 2

now  $arr[st.top()] > curElement \Rightarrow ht = arr[st.top()] \ \& \ st.pop() \uparrow$

width = i - st.top() - 1 = 1 ∴ area = 6 \* 1 = 6      maxArea = 6.

& push(i)<sup>4</sup>





$\rightarrow$   $[2, 1, 5, 6, 2, 3]$   $[1, 2]$   $area = 10$   $maxArea = 10$   
 $i = 4$   $now\ arr[st.top()] > currentElement \Rightarrow ht = arr[st.top()] \ \& \ st.pop()$   
 $width = i - st.top() - 1 = 2 \therefore area = 5 \times 2 = 10$   $maxArea = 10$   
 $\& \ push(i)$

$\rightarrow$   $[2, 1, 5, 6, 2, 3]$   $[1, 4]$   $area = 0$   $maxArea = 10$   
 $i = 5$   $now\ arr[st.top()] > currentElement \Rightarrow false \therefore push(i)$

$\Rightarrow$  Last iteration to pop stack  $\Rightarrow i = 6$

$\rightarrow$   $[2, 1, 5, 6, 2, 3]$   $[1, 4, 5]$   $area = 3$   $maxArea = 10$   
 $ht = arr[st.top()] \ \& \ pop()$   $\& \ as\ stack\ is\ not\ empty$   
 $width = i - st.top() - 1 = 1 \therefore area = 3 \times 1 = 3$   $maxArea = 10$

$\rightarrow$   $[2, 1, 5, 6, 2, 3]$   $[1, 4]$   $area = 3$   $maxArea = 10$   
 $ht = arr[st.top()] \ \& \ pop()$   $\& \ as\ stack\ is\ not\ empty$   
 $width = i - st.top() - 1 = 4 \therefore area = 2 \times 4 = 8$   $maxArea = 10$

<sup>0 1 2 3 4 5</sup>  
 → [2, 1, 5, 6, 2, 3] [1, ] area = 6 maxArea = 10  
 ht = arr[st.top()] & pop() & as stack is empty  
 width = i = 6 ⇒ ∴ area = 1 × 6 = 6 maxArea = 10  
 ∴ As stack is empty return maxArea = 10.

Code →

Tc → O(n)

Sc → O(n)

```

1  class Solution {
2  public:
3      int largestRectangleArea(vector<int>& heights) {
4          stack < int > st;
5          int maxArea = 0;
6          int n = heights.size();
7
8          for (int i = 0; i <= n; i++) {
9
10             while (!st.empty() && (i == n || heights[st.top()] >= heights[i])) {
11
12                 int height = heights[st.top()];
13                 st.pop();
14                 int width;
15                 if (st.empty()){
16                     width = i;
17                 } else {
18                     width = i - st.top() - 1;
19                 }
20
21                 int area = width*height;
22                 maxArea = max(maxArea, area);
23             }
24             st.push(i);
25         }
26         return maxArea;
27     }
28 };
29
30
  
```

## ⑫ Sliding Window Maximum →

- process first 'k' elements before pushing into result arr.
- if  $dq.front() == i - k$  then pop-front (out of boundary case)
- if  $nums[dq.back()] < nums[i]$  then pop-back  
(meaningless to store smaller elements in window)
- if  $i \geq k - 1$  then push  $nums[dq.front()]$

Eg  $nums = [1, 3, -1, -3, 5, 3, 6, 7]$   $k = 3$   $res = [3, 3, 5, 5, 6, 7]$

	$nums$	$deque$	$res$
→	$[1, 3, -1, -3, 5, 3, 6, 7]$ 0 1 2 3 4 5 6 7	<hr/> <hr/>	$[ \quad ]$
$i=0$	$[1, 3, -1, -3, 5, 3, 6, 7]$ 0 1 2 3 4 5 6 7	<hr/> <hr/> 0	$[ \quad ]$
$i=1$	$[1, 3, -1, -3, 5, 3, 6, 7]$ 0 1 2 3 4 5 6 7 → $dq.front == i - k \rightarrow false$ $nums[0] < nums[i]$ ∴ pop back & push i	<hr/> <hr/> 0 <hr/> <hr/> 1	$[ \quad ]$
$i=2$	$[1, 3, -1, -3, 5, 3, 6, 7]$ 0 1 2 3 4 5 6 7 → $dq.front == i - k \rightarrow false$ $nums[1] < nums[i]$ ∴ false & push i	<hr/> <hr/> 1, 2	$[ 3 \quad ]$ ↑ 3

→ as  $i \geq k - 1$   
push  $nums[dq.front()]$  i.e. 3 into res



$i=3$   $[1, 3, -1, -3, 5, 3, 6, 7]$

$\rightarrow dq.front == i-k \rightarrow false$

$num[2] < num[i]$

$\therefore false$  & push  $i$

1, 2, 3

$[3, 3]$

$\rightarrow as i \geq k-1$

push  $num[dq.front()]$  i.e. 3  
into res

$i=4$   $[1, 3, -1, -3, 5, 3, 6, 7]$

$\rightarrow dq.front == i-k$   $\therefore pop-front$

$num[3] < num[i]$   $\therefore pop-back$

$num[2] < num[i]$   $\therefore pop-back$

& push( $i$ )

order & pop  
1, 2, 3, 4

$[3, 3, 5]$

$\rightarrow as i \geq k-1$

push  $num[dq.front()]$  i.e. 5  
into res

$i=5$   $[1, 3, -1, -3, 5, 3, 6, 7]$

$\rightarrow dq.front == i-k \rightarrow false$

$num[4] < num[i]$

$\therefore false$  & push( $i$ )

order & pop  
4, 5

$[3, 3, 5, 5]$

$\rightarrow as i \geq k-1$

push  $num[dq.front()]$  i.e. 5  
into res

$i=6$   $[1, 3, -1, -3, 5, 3, 6, 7]$

$\rightarrow dq.front == i-k \rightarrow false$

$num[5] < num[i]$   $\therefore pop-back$

$num[4] < num[i]$   $\therefore pop-back$

& push

order & pop  
4, 5, 6

$[3, 3, 5, 5, 6]$

$\rightarrow as i \geq k-1$

push  $num[dq.front()]$  i.e. 6  
into res

$i=7$  [1, 3, -1, -3, 5, 3, 6, 7]

order of pop  
① 6, 7

[3, 3, 5, 5, 6, 7]

→  $dq.front() == i - k \rightarrow false$

$nums[6] < nums[i] \therefore pop\_back$

& push(i)

→ as  $i \geq k - 1$

push  $nums[dq.front()]$  into res

code →

$T_c \rightarrow O(N)$

$S_c \rightarrow O(K)$

```
1 class Solution {
2 public:
3     vector<int> maxSlidingWindow(vector<int>& nums, int k) {
4         deque<int> dq;
5         vector<int> ans;
6         for (int i = 0; i < nums.size(); i++) {
7
8             if (!dq.empty() && dq.front() == i - k)
9                 dq.pop_front();
10
11             while (!dq.empty() && nums[dq.back()] < nums[i])
12                 dq.pop_back();
13
14             dq.push_back(i);
15
16             if (i >= k - 1)
17                 ans.push_back(nums[dq.front()]);
18         }
19         return ans;
20     }
21 };
```



# Linked List

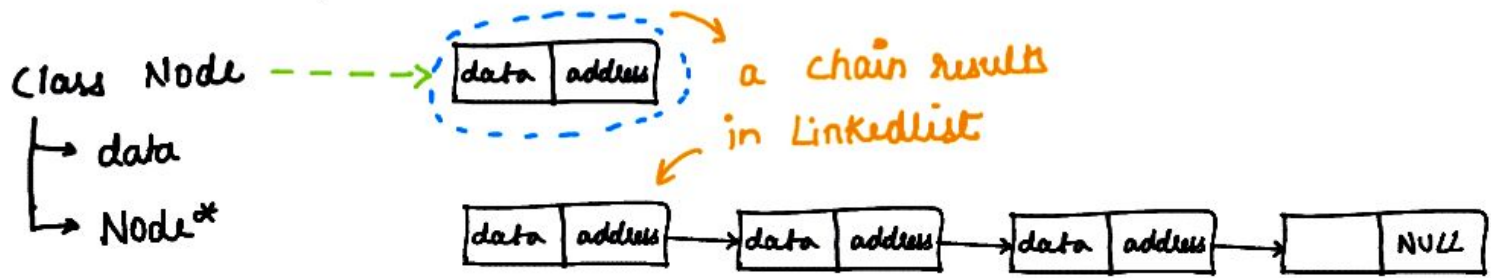
- Karun Karthik

## Contents →

0. Introduction
1. Reverse a Linked List
2. Middle of Linked List
3. Delete node in a Linked List
4. Merge two sorted Lists
5. Add two numbers
6. Add two numbers II
7. Linked List Cycle
8. Linked List Cycle II
9. Remove Nth node from End of List
10. Palindrome Linked List
11. Remove duplicates from sorted List
12. Swapping nodes in Linked List
13. Odd Even Linked List
14. Swap Nodes in Pairs
15. Copy list with Random Pointer
16. Reverse Nodes in K-group
17. Design Linked List
18. Sort List

# Linked List

LinkedList is linear data structure, which consists of a group of nodes in a sequence.



## Advantages

1. Dynamic nature
2. Optimal insertion & deletion
3. Stacks and queues can be easily implemented
4. No memory wastage

## Disadvantages

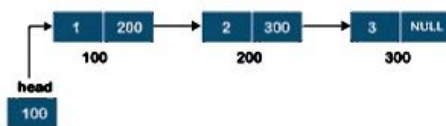
1. More memory usage due to address pointer.
2. Slow traversal compared to arrays.
3. No reverse traversal in singly linked list
4. No random access.

## Real life Applications

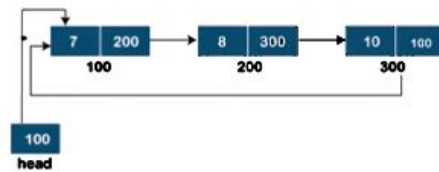
1. Previous & next page in browser
2. Image Viewer
3. Music player

## Types

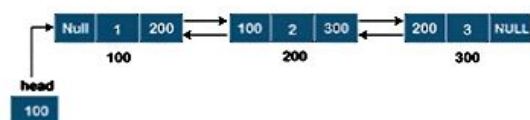
### 1. Singly linkedlist



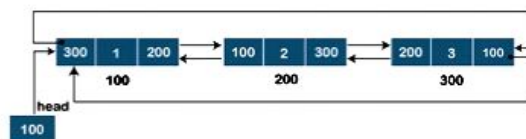
### 3. Circular linkedlist



### 2. Doubly linkedlist



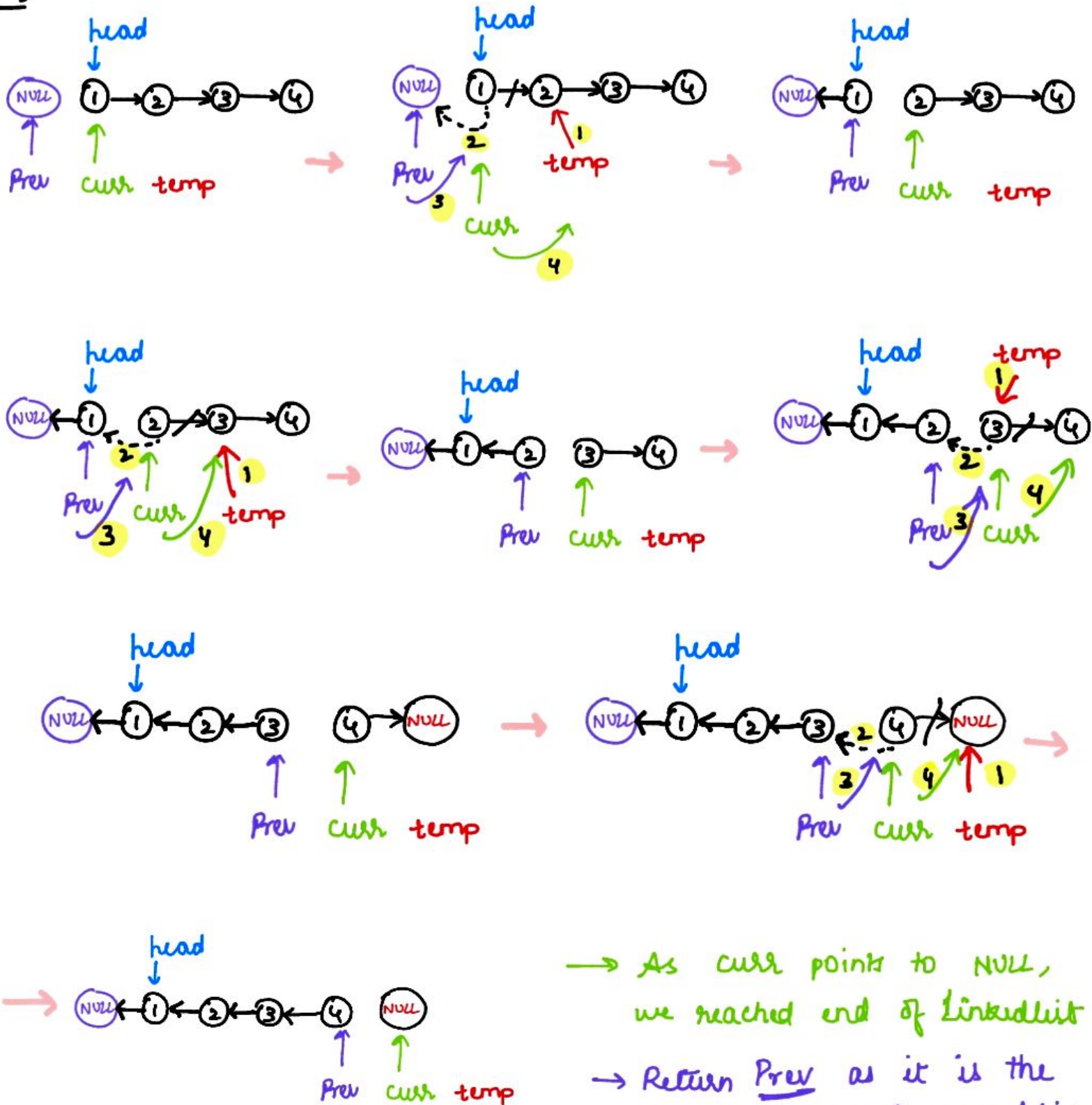
### 4. Doubly circular linkedlist



① Reverse a linkedlist → given a linkedlist, return reversed list.

Eg  $1 \rightarrow 2 \rightarrow 3 \rightarrow 4 \Rightarrow 4 \rightarrow 3 \rightarrow 2 \rightarrow 1$

Sol)



→ As `curr` points to NULL, we reached end of linkedlist

→ Return prev as it is the starting pointer of reversed list.

# Recursive →



curr, prev

(1, NULL):

newNode = 1 → next = 2

1 → next = NULL

call(2, 1)

curr, prev

(2, 1):

newNode = 2 → next = 3

2 → next = 1

call(3, 2)

curr, prev

(3, 2):

newNode = 3 → next = 4

3 → next = 2

call(4, 3)

curr, prev

(4, 3):

newNode = 4 → next = NULL

4 → next = 3

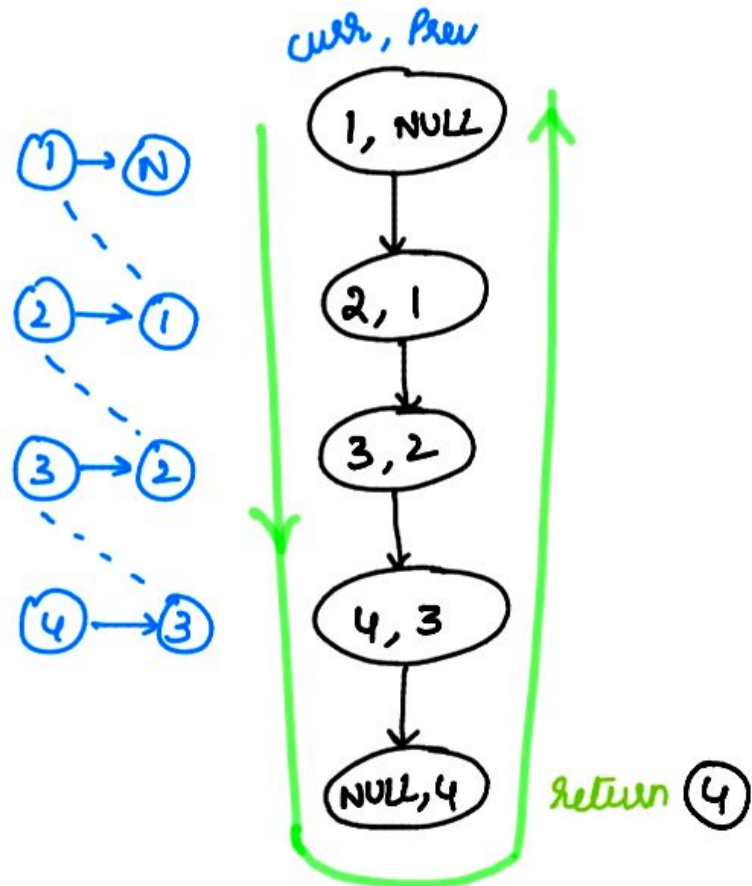
call(NULL, 4)

curr, prev

(NULL, 4):

as curr == NULL

return prev



func(curr, prev):

if curr == NULL

return prev

newNode = curr → next

curr → next = prev

recursively call newNode & curr as  
as curr prev



Code →

$T_c \rightarrow O(n)$

$S_c \rightarrow O(1)$

```
1
2 // Iterative ->
3 class Solution {
4 public:
5     ListNode* reverseList(ListNode* head) {
6         ListNode *prev = NULL, *curr=head, *temp;
7         while(curr){
8             temp = curr->next;
9             curr->next = prev;
10            prev = curr;
11            curr = temp;
12        }
13        return prev;
14    }
15 };
16
17
18 // Recursive ->
19 class Solution {
20 public:
21     ListNode* reverseLinker(ListNode* curr, ListNode* prev) {
22         if(curr==NULL)
23             return prev;
24         ListNode* newNode = curr->next;
25         curr->next = prev;
26         return helper(newNode, curr);
27     }
28
29     ListNode* reverseLinker(ListNode* head) {
30         return helper (head, NULL);
31     }
32 };
```



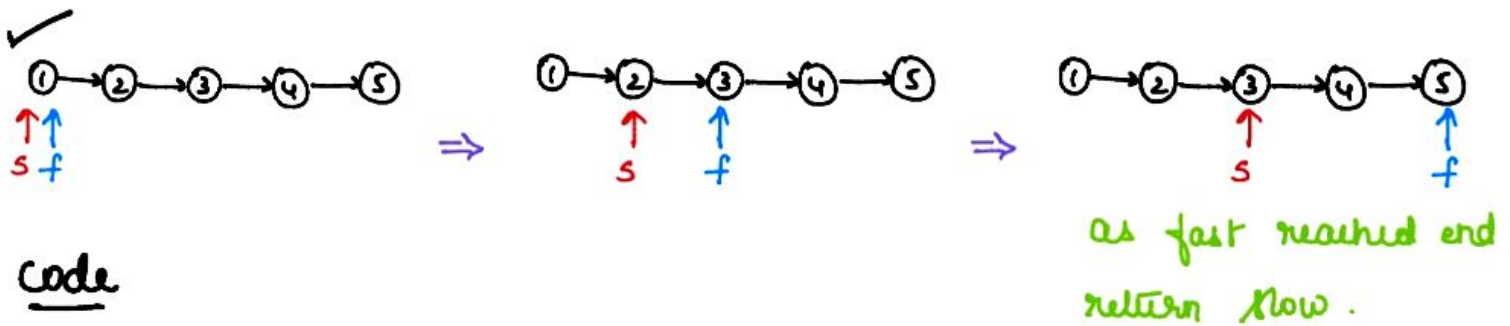
② Middle of LinkedList → Given the head, return middle node.

Approach 1 → Traverse the list & find no of nodes & return mid

Approach 2 → use 2 pointers → slow (moves by 1) } By time  
fast (moves by 2) } fast reaches end, slow points to the mid.

Eg ① → ② → ③ → ④ → ⑤

Res → ③ → ④ → ⑤



code

```
1 class Solution {
2 public:
3     ListNode* middleNode(ListNode* head) {
4         if(head == NULL)
5             return head;
6         ListNode* slow = head, *fast = head;
7
8         // Traverse the LinkedList
9         while(fast != NULL && fast -> next != NULL)
10        {
11            slow = slow -> next;
12            fast = fast -> next -> next;
13        }
14
15        return slow;
16    }
17 };
```

Tc →  $O(n)$

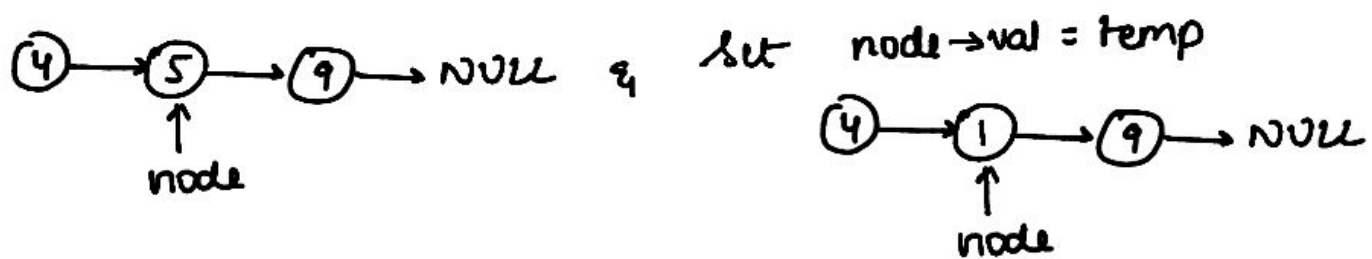
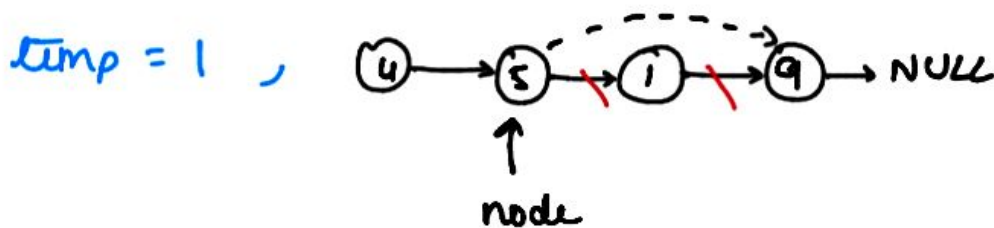
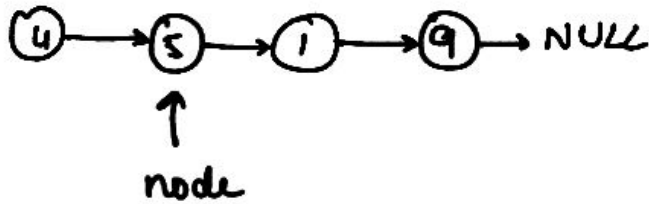
Sc →  $O(1)$

### ③ Delete node in a linkedlist →

Given a linkedlist's node, delete it.

- copy node's next node's val into a temp variable
- skip the node → next node
- copy the temp variable's value into the node.

Eg

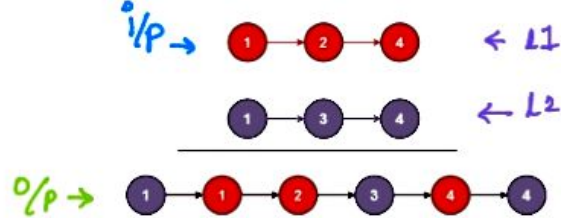


code →

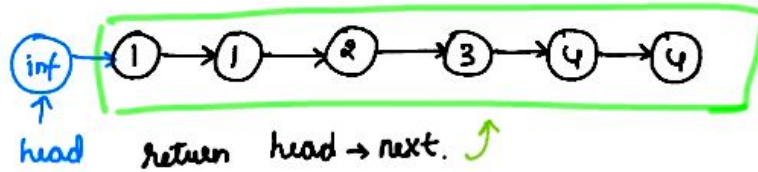
```
1 class Solution {
2 public:
3     void deleteNode(ListNode* node) {
4         int val = node->next->val;
5         node->next = node->next->next;
6         node->val = val;
7     }
8 };
```

Tc → O(1)  
Sc → O(1)

#### ④ Merge two sorted lists →



→ Take a dummyNode & chain the next node which contains smaller value of  $L_1$  &  $L_2$



Code →

$T_c \rightarrow O(m+n)$

$S_c \rightarrow O(m+n)$

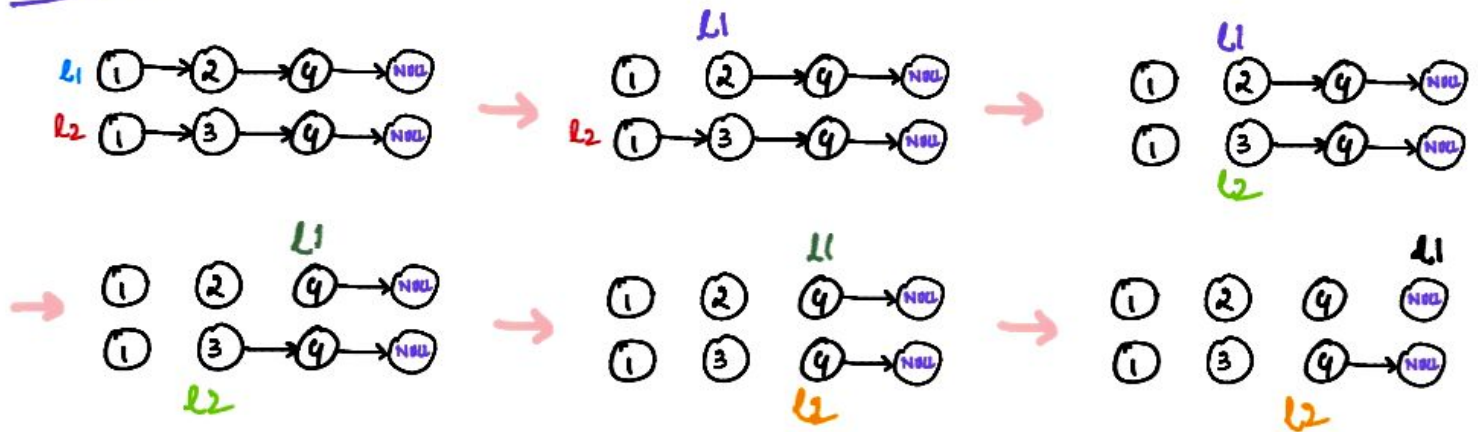
```

1  class Solution {
2  public:
3      ListNode* mergeTwoLists(ListNode* l1, ListNode* l2) {
4
5          if( l1 == NULL ) return l2;
6          if( l2 == NULL ) return l1;
7
8          ListNode* dummy = new ListNode(-101);
9          ListNode* head = dummy;
10
11         // Traverse the lists
12         while( l1 != NULL && l2 != NULL )
13         {
14             if( l1->val < l2->val )
15             {
16                 ListNode* newnode = new ListNode(l1->val);
17                 dummy->next = newnode;
18                 l1 = l1->next;
19             }
20             else
21             {
22                 ListNode* newnode = new ListNode(l2->val);
23                 dummy->next = newnode;
24                 l2 = l2->next;
25             }
26             dummy = dummy->next;
27         }
28
29         /* If a particular list is NULL, the directly chain
30         the other */
31         if(l1!=NULL) dummy->next = l1;
32         if(l2!=NULL) dummy->next = l2;
33
34         return head->next;
35     }
36 };
  
```

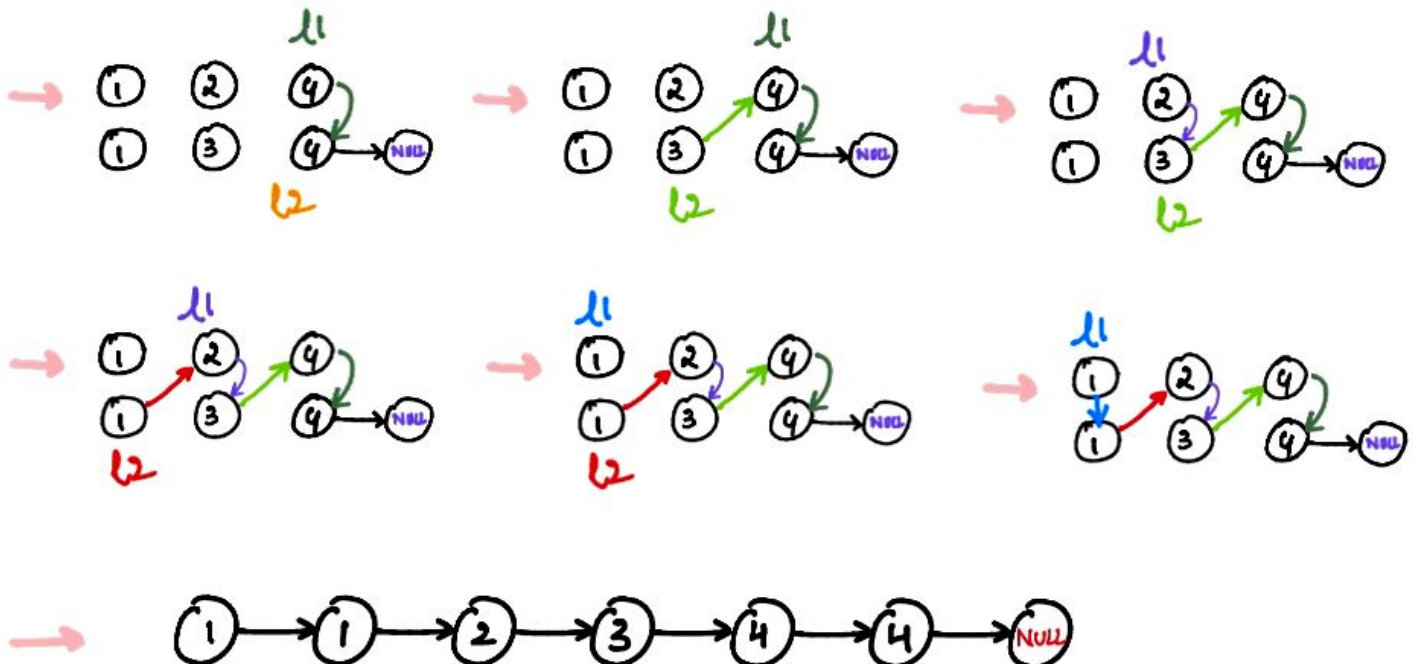
## Recursive Code →

```
class Solution {  
public:  
    ListNode* mergeTwoLists(ListNode* l1, ListNode* l2) {  
  
        if (l1 == NULL) return l2;  
        if (l2 == NULL) return l1;  
  
        // compare the starting values and link the nodes  
        if (l1->val <= l2->val) {  
            l1->next = mergeTwoLists(l1->next, l2);  
            return l1;  
        } else {  
            l2->next = mergeTwoLists(l1, l2->next);  
            return l2;  
        }  
    }  
};
```

## Dry Run



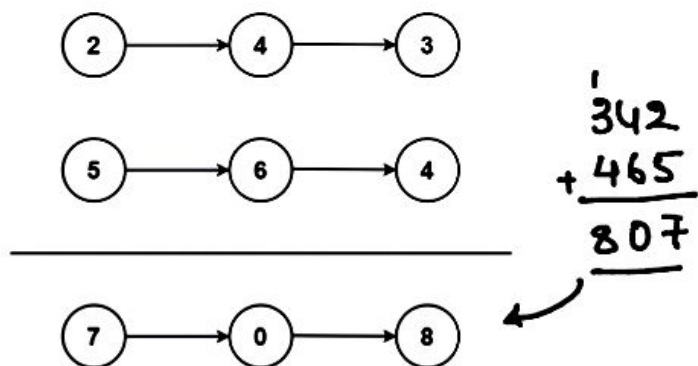
As l1 is null, return l2





## ⑤ Add two Numbers

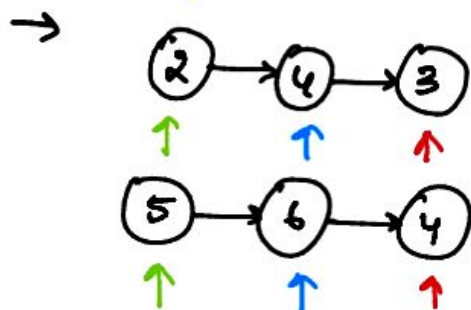
Given 2 lists in reverse order, add them and return the sum.



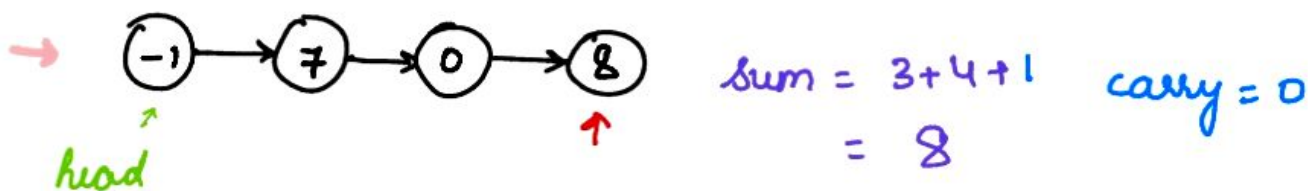
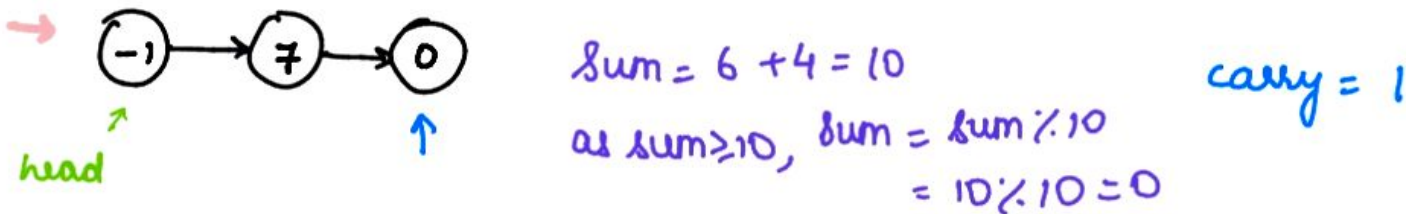
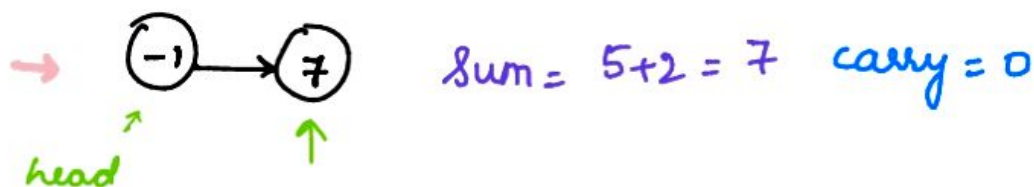
$$T.C \rightarrow O(m+n)$$

$$S.C \rightarrow O(\max(m,n))$$

Initially,



- 1) traverse both list simultaneously & if  $\text{sum} \geq 10$ , then set  $\text{carry} = 1$
- 2) add both values + carry
- 3) Create newNode with this value



code

```
1  class Solution {
2  public:
3
4      ListNode* addTwoNumbers(ListNode* l1, ListNode* l2) {
5
6          ListNode* dummyNode;
7          ListNode* head;
8          dummyNode = head = new ListNode(-1);
9          if(!l1)
10             return l2;
11          if(!l2)
12             return l1;
13
14          int carry = 0;
15
16          while(l1 || l2){
17              int firstVal = l1 ? l1->val : 0;
18              int secondVal = l2 ? l2->val : 0;
19
20              int total = firstVal + secondVal + carry;
21              carry = total / 10;
22              total = total % 10;
23
24              ListNode* newNode = new ListNode(total);
25              dummyNode->next = newNode;
26
27              dummyNode = dummyNode->next;
28
29              l1 = l1 ? l1->next : l1;
30              l2 = l2 ? l2->next : l2;
31          }
32
33          if(carry)
34              dummyNode->next = new ListNode(1);
35
36          return head->next;
37      }
38  };
```

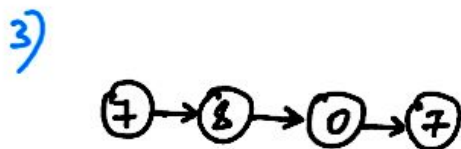
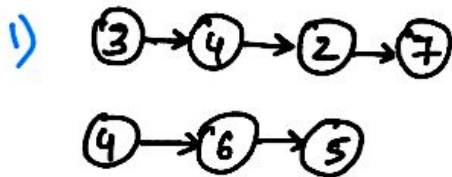
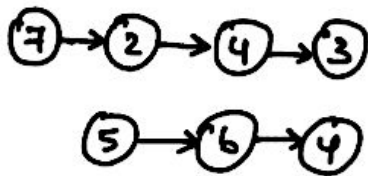
## ⑥ Add two numbers II

→ Problem solving approach is same as previous problem

→ Points to note:

1. Reverse both the lists
2. Add them
3. Reverse the result

Eg →



Result ↑

Code →

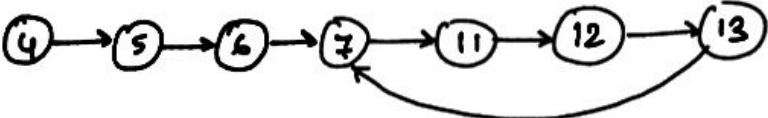
```
1 class Solution {
2 public:
3     ListNode* reverseList(ListNode* head) {
4         ListNode* prev = NULL;
5         ListNode* curr = head;
6         ListNode* temp = NULL;
7         while(curr != NULL)
8         {
9             temp = curr->next;
10            curr->next = prev;
11            prev = curr;
12            curr = temp;
13        }
14        return prev;
15    }
16
17    ListNode* addTwoNumbers(ListNode* l1, ListNode* l2) {
18        l1 = reverseList(l1); // O(n)
19        l2 = reverseList(l2); // O(n)
20        ListNode* dummyNode;
21        ListNode* head;
22        dummyNode = head = new ListNode(-1);
23        if(!l1)
24            return l2;
25        if(!l2)
26            return l1;
27
28        int carry = 0;
29
30        while(l1 || l2){
31            int firstVal = l1 ? l1->val : 0;
32            int secondVal = l2 ? l2->val : 0;
33
34            int total = firstVal + secondVal + carry;
35            carry = total / 10;
36            total = total % 10;
37
38            ListNode* newNode = new ListNode(total);
39            dummyNode->next = newNode;
40
41            dummyNode = dummyNode->next;
42
43            l1 = l1 ? l1->next : l1;
44            l2 = l2 ? l2->next : l2;
45        }
46
47        if(carry)
48            dummyNode->next = new ListNode(1);
49
50        return reverseList(head->next); // O(max(m,n))
51    }
52};
```

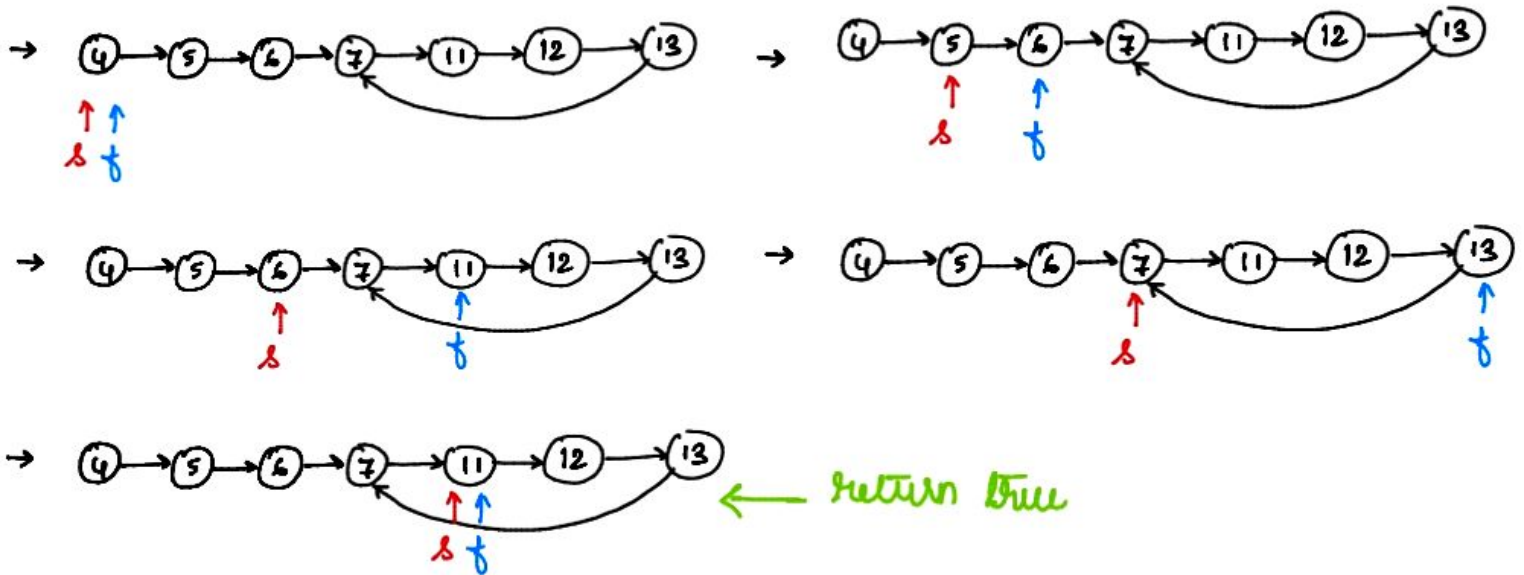
## ⑦ Linked List Cycle →

TC →  $O(n)$   
SC →  $O(1)$

Approach - 1 → Create a set of nodes & insert every node into it, if already exist then return True else false.

Approach - 2 → using fast & slow pointer TC →  $O(n)$  SC →  $O(1)$

Eg  keep iterating till  
fast → next && fast → next → next exist.

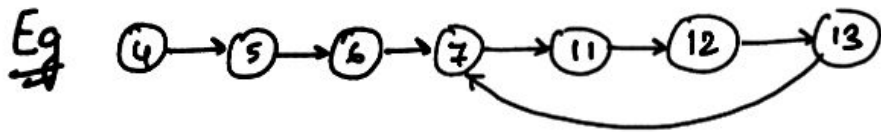


code →

```
1 class Solution {
2 public:
3     bool hasCycle(ListNode *head) {
4         if(head==NULL) return false;
5         ListNode *fast = head, *slow = head;
6         while(fast->next!=NULL && fast->next->next!=NULL)
7         {
8             fast=fast->next->next;
9             slow=slow->next;
10            if(fast==slow) return true;
11        }
12        return false;
13    }
14 };
```

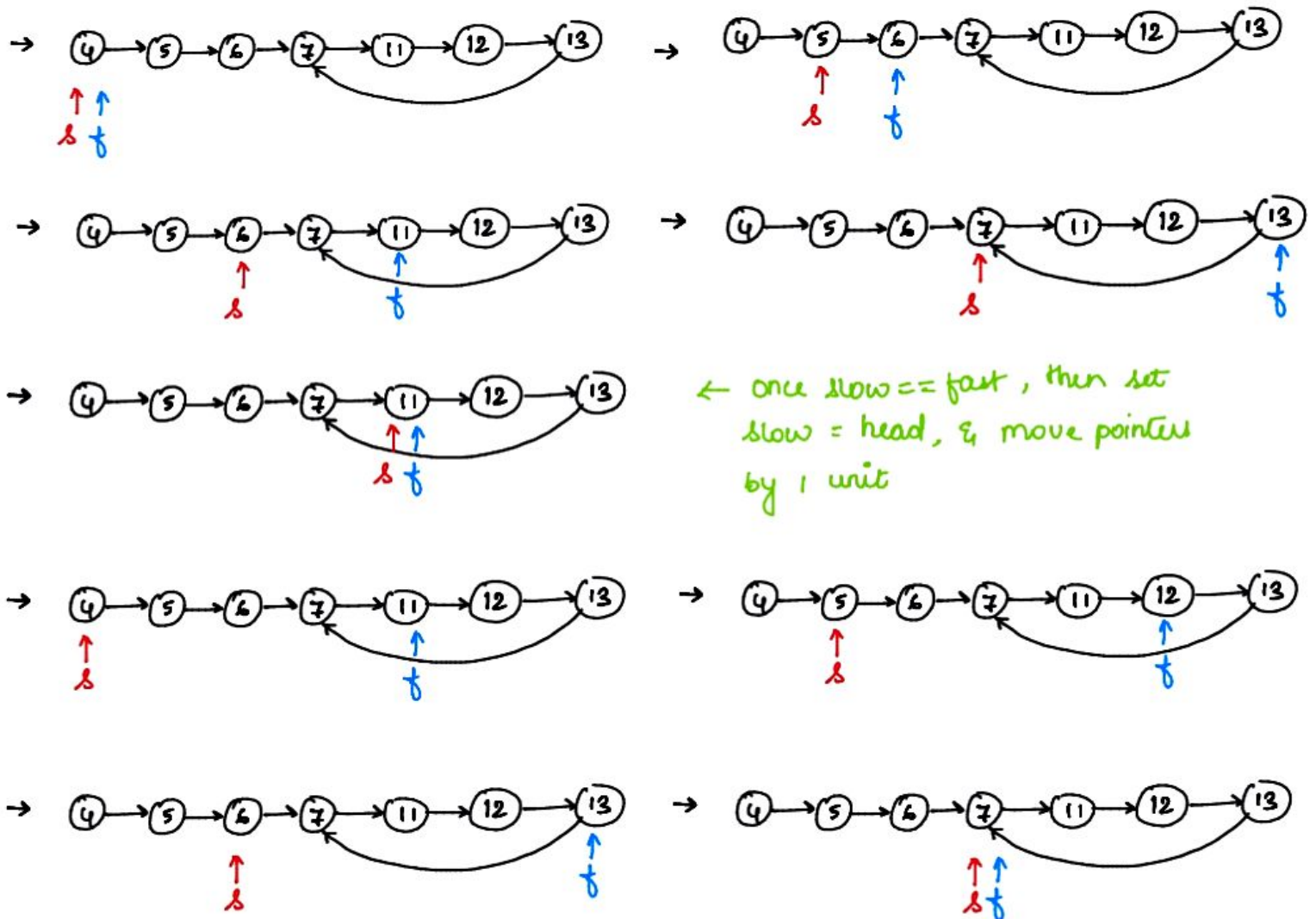


(8) Linked list cycle II  $\rightarrow$  returns the node where cycle begins.



result = 7

keep iterating while  $\text{fast} \rightarrow \text{next}$  &&  $\text{fast} \rightarrow \text{next} \rightarrow \text{next}$  exist.



$\rightarrow$  when  $\text{slow} == \text{fast}$ , it denotes the node where cycle begins.

return slow;  $\Rightarrow$  7

code →

```
1 class Solution {
2 public:
3     ListNode *detectCycle(ListNode *head) {
4         if(head==NULL) return NULL;
5         ListNode *fast = head, *slow = head;
6         while(fast->next!=NULL && fast->next->next!=NULL)
7         {
8             fast = fast->next->next;
9             slow = slow->next;
10            if(fast == slow)
11            {
12                slow = head;
13                while (slow != fast)
14                {
15                    slow = slow->next;
16                    fast = fast->next;
17                }
18                return slow;
19            }
20        }
21        return NULL;
22    }
23 };
```

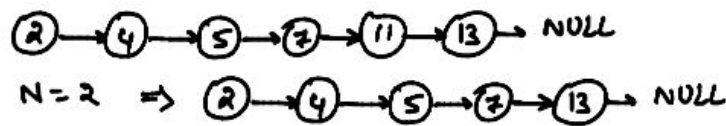
$$T_c \rightarrow O(n) + O(n) = O(2n) = \underline{\underline{O(n)}}$$

\* worst case when its a loop



$$S.C \rightarrow O(1)$$

## ⑨ Remove N<sup>th</sup> node from End of List



Approach - 1 → find length of list L, delete  $L - N + 1$ <sup>th</sup> node

Approach - 2 → a) Reverse b) Delete N<sup>th</sup> node c) Reverse

code →

TC →  $O(n)$

SC →  $O(1)$

```
1 class Solution {
2 public:
3     ListNode* reverseList(ListNode* head) {
4         ListNode *prev = NULL, *curr = head, *temp;
5         while(curr){
6             temp = curr->next;
7             curr->next = prev;
8             prev = curr;
9             curr = temp;
10        }
11        return prev;
12    }
13
14    ListNode* removeNthFromEnd(ListNode* head, int n) {
15        ListNode *dummy = new ListNode(-1);
16        dummy->next = reverseList(head);
17        head = dummy;
18        ListNode *curr = head;
19        ListNode *prev = NULL;
20        // Iteration
21        for(int i=0; i<n; i++)
22        {
23            prev = curr;
24            curr = curr->next;
25        }
26        // Deletion
27        prev->next = curr->next;
28        return reverseList(head->next);
29    }
30 };
```

## ⑩ Palindrome LinkedList

Approach - 1 → create a copy of list & reverse it. Compare value by value.  
If all are equal **true** else **false**.

Approach - 2 → Reach middle node & return the remaining list as new list. Reverse the new list & compare its value by value.

code →

Tc →  $O(n)$

Sc →  $O(1)$

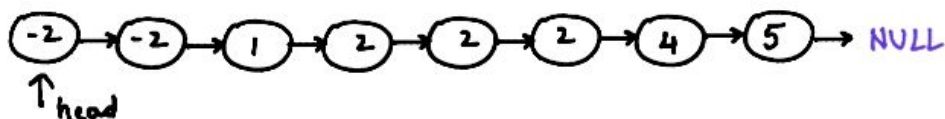
```
1 class Solution {
2 public:
3     ListNode* midNode(ListNode* head)
4     {
5         ListNode *fast = head, *slow = head;
6         while(fast->next!=NULL and fast->next->next!=NULL){
7             fast=fast->next->next;
8             slow = slow->next;
9         }
10        return slow;
11    }
12    ListNode* reverseList(ListNode* head) {
13        ListNode *prev = NULL, *curr = head, *temp;
14        while(curr!=NULL)
15        {
16            temp=curr->next;
17            curr->next = prev;
18            prev = curr;
19            curr = temp;
20        }
21        return prev;
22    }
23    bool compare(ListNode* l1, ListNode* l2)
24    {
25        while(l1!=NULL && l2!=NULL)
26        {
27            if(l1->val!=l2->val) return false;
28            l1 = l1->next;
29            l2 = l2->next;
30        }
31        return true;
32    }
33    bool isPalindrome(ListNode* head) {
34        if(head==NULL) return false;
35        if(head->next == NULL) return true;
36        ListNode *mid = midNode(head);
37        ListNode *l2 = mid->next;
38        mid->next = NULL;
39        l2 = reverseList(l2);
40        return compare(head, l2);
41    }
42 };
```



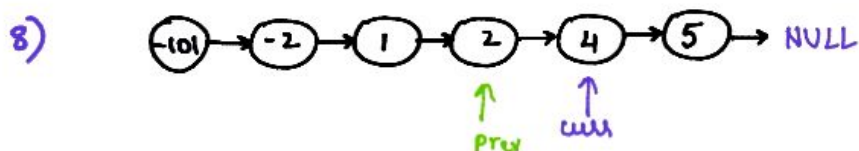
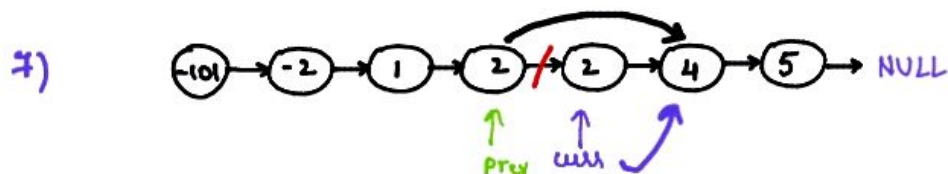
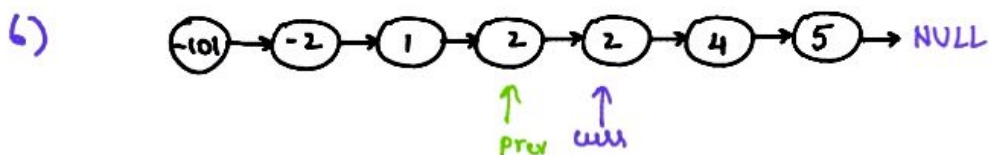
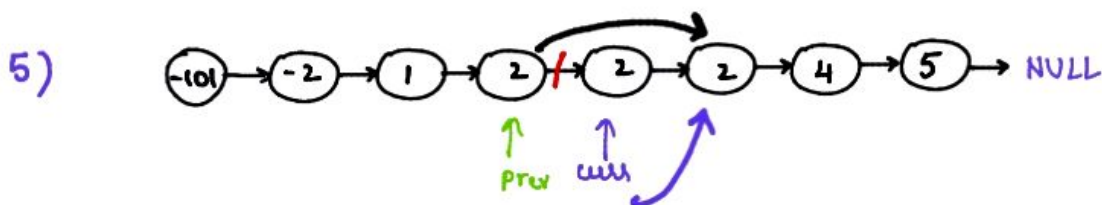
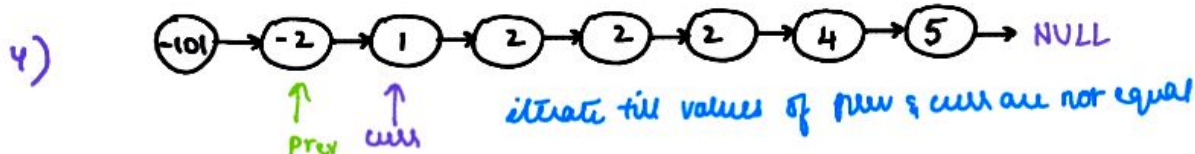
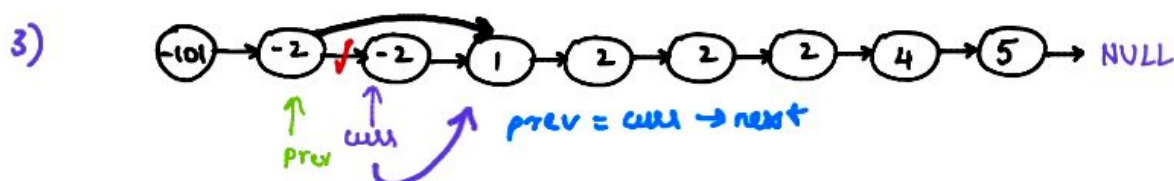
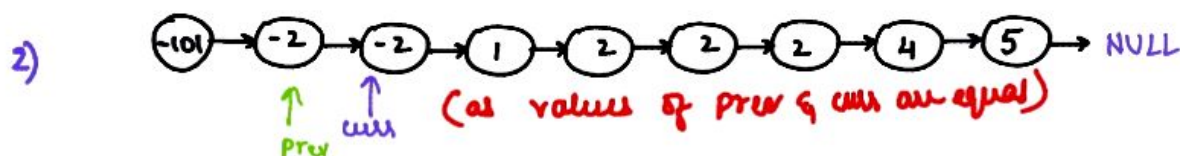
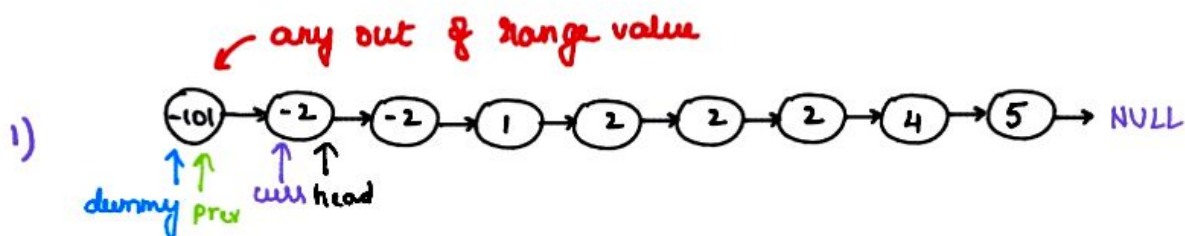
# (11) Remove duplicates from sorted list →

Given a linkedlist, return linkedlist without duplicates.

Ex



⇒



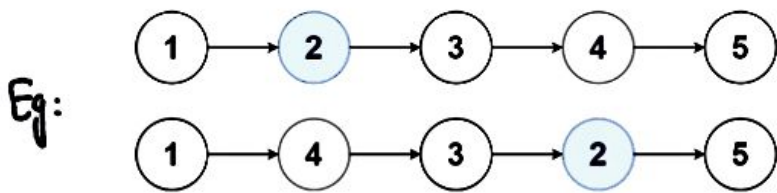
code →

```
1  class Solution {
2  public:
3      ListNode* deleteDuplicates(ListNode* head) {
4          ListNode* dummy = new ListNode(101);
5          dummy->next = head;
6          ListNode* curr = head;
7          ListNode* prev = dummy;
8          while(curr!=NULL)
9          {
10             if(curr->val==prev->val){
11                 prev->next = curr->next;
12                 curr = curr->next;
13             } else {
14                 prev = curr;
15                 curr = curr->next;
16             }
17         }
18         return dummy->next;
19     }
20 };
21
22 // Another approach
23
24
25 ListNode* deleteDuplicates(ListNode* head) {
26     if(head==NULL || head->next==NULL) return head;
27     ListNode *curr = head;
28     while(curr->next!=NULL){
29         if(curr->val == curr->next->val){
30             curr->next = curr->next->next;
31         } else {
32             curr = curr->next;
33         }
34     }
35     return head;
36 }
```

$T_c \rightarrow O(n)$

$S_c \rightarrow O(1)$

## Q2) Swapping Nodes in Linked list →

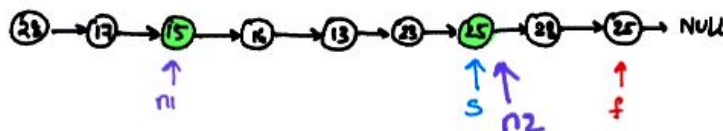
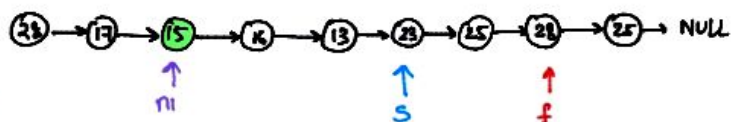
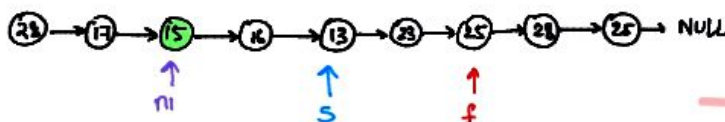
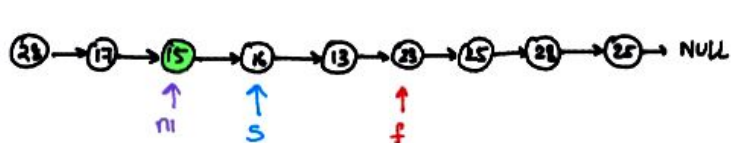
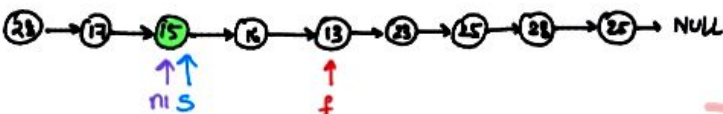
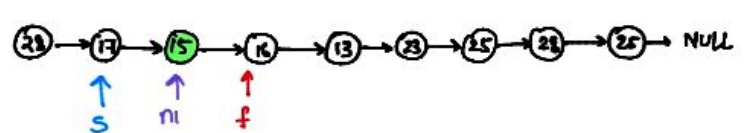
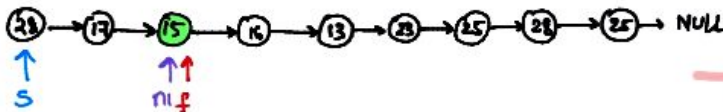
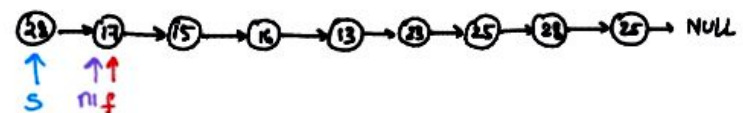
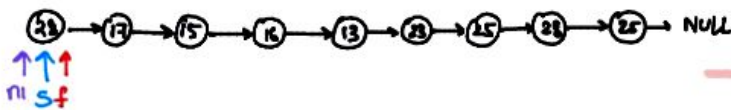
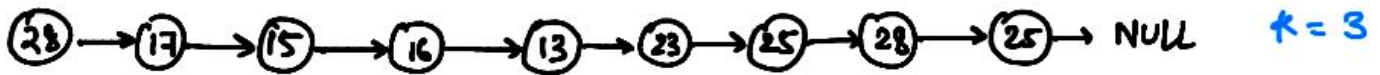


$k=2$

Given a linkedlist swap  $k^{\text{th}}$  node from both ends.

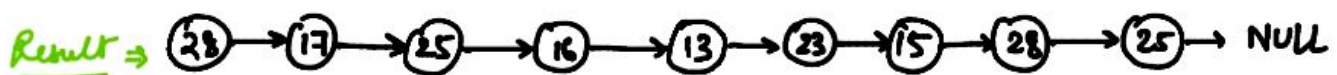
Eg

\* for  $k-1$  times iterate  $f$  & mark  $n1$ , then iterate  $s$  &  $f$  till  $f$  is not NULL, once null mark  $s$  as  $n2$ .  
(as it is 1 indexed) swap ( $n1, n2$ )



( $f \rightarrow \text{next} = \text{NULL}$ , so  $s$  is  $n2$ )

swap (15, 25)



code →

```
1  class Solution {
2  public:
3      ListNode* swapNodes(ListNode* head, int k) {
4          ListNode *slow = head, *fast = head, *n1 = head;
5          // finding n1
6          for(int i=0; i<k-1; i++){
7              fast = fast->next;
8              n1 = fast;
9          }
10         // finding n2 (i.e slow)
11         while(fast->next!=NULL){
12             fast = fast->next;
13             slow = slow->next;
14         }
15         // swapping
16         int n1_val = n1->val;
17         n1->val = slow->val;
18         slow->val = n1_val;
19         return head;
20     }
21 };
```

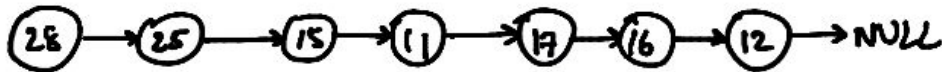
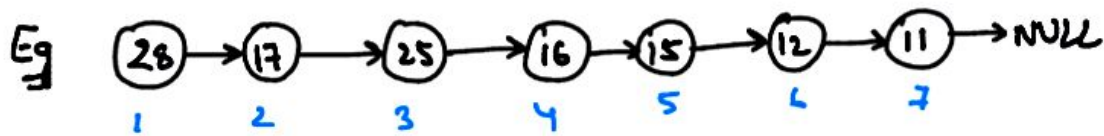
$T_c \rightarrow O(n)$

$SC \rightarrow O(1)$

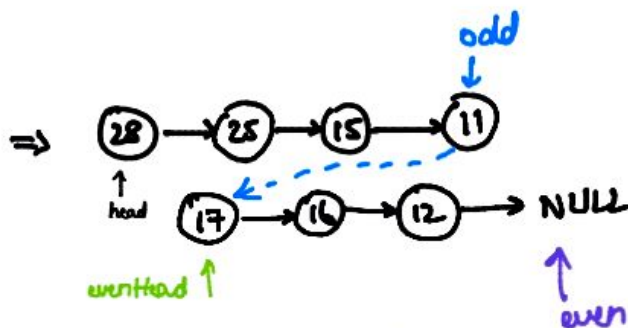
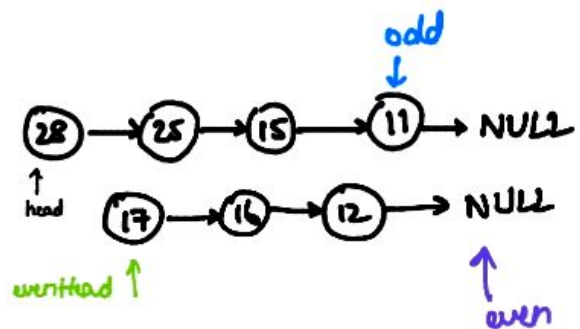
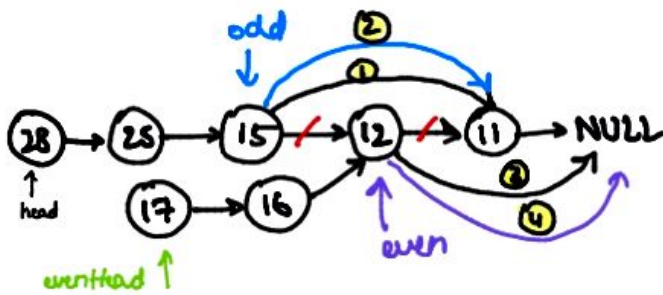
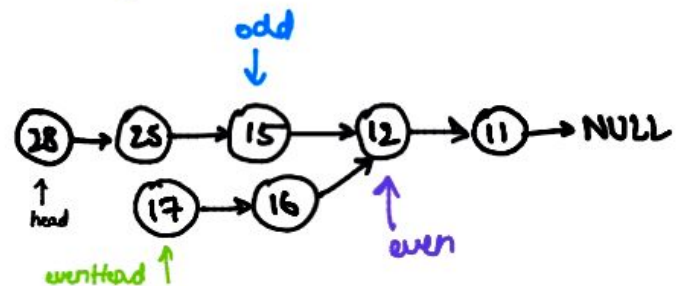
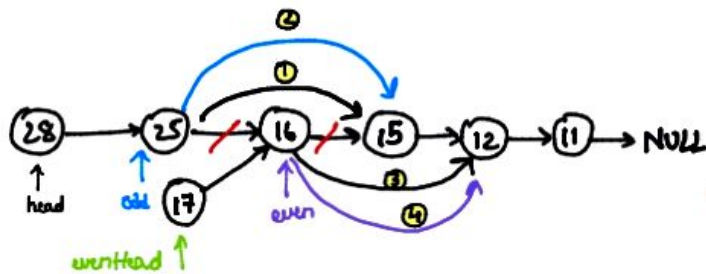
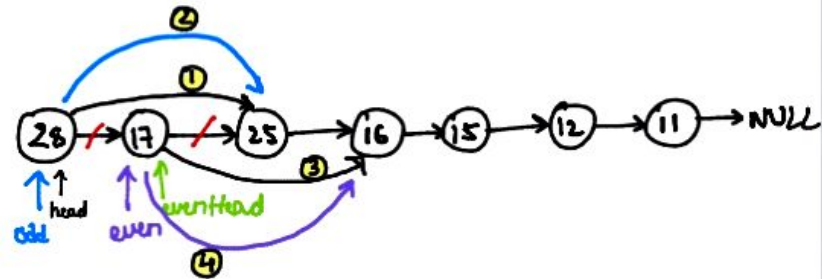
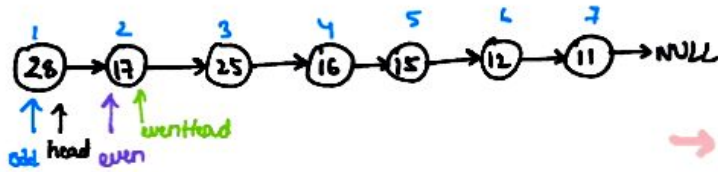


### 13) Odd Even Linked List →

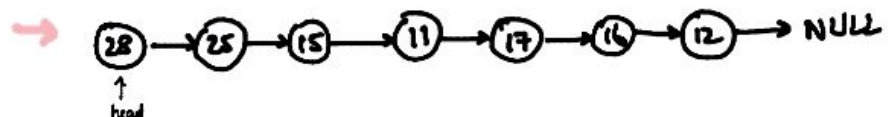
Given a linkedlist group all odd indices nodes followed by even nodes



⇒



odd → next = evenhead



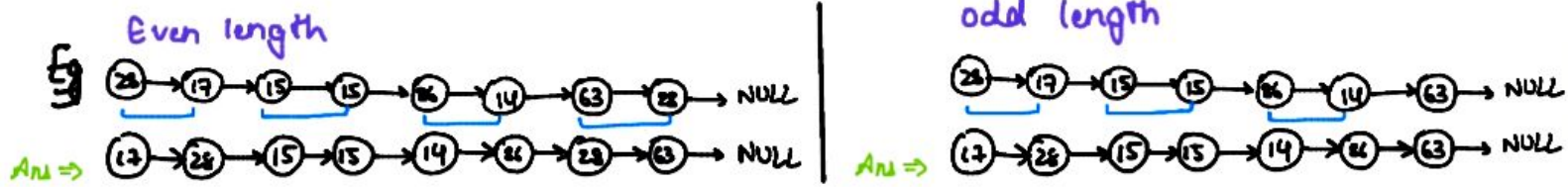
code →

```
1  class Solution {
2  public:
3      ListNode* oddEvenList(ListNode* head) {
4          if(!head) return NULL;
5
6          ListNode *even = head->next;
7          ListNode *odd = head;
8          ListNode *evenHead = even;
9
10         while(even && even->next){
11             odd->next=even->next;
12             odd=odd->next;
13             even->next=odd->next;
14             even=even->next;
15         }
16
17         // like odd and even lists
18         odd->next = evenHead;
19         return head;
20     }
21 };
```

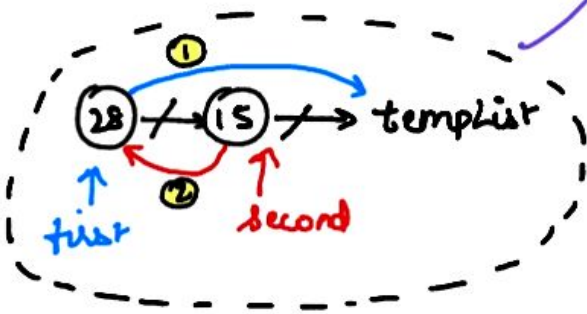
$T_c \rightarrow O(n)$

$Sc \rightarrow O(1)$

(14) Swap Nodes in Pairs → Given a linkedlist swap adjacent nodes.



Consider for 1st pair,



Logic

- templist = second → next
- first → next = templist
- second → next = first

Solve recursively for all pairs.

Code →

```

1 class Solution {
2 public:
3     ListNode* SwapAdjacentNodes(ListNode* head)
4     {
5         if(head==NULL || head->next==NULL) return head;
6         ListNode *first = head;
7         ListNode *second = head->next;
8         // start logic
9         ListNode *tempList = SwapAdjacentNodes(second->next);
10        first->next = tempList;
11        second->next = first;
12        return second;
13    }
14    ListNode* swapPairs(ListNode* head) {
15        return SwapAdjacentNodes(head);
16    }
17 };

```

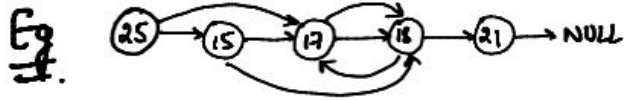
Tc →  $O(N)$

Sc →  $O(1)$

Recursive  
stack →  $O(N/2)$   
≈  $O(N)$



15) Copy list with random pointer → given a list, clone & return.

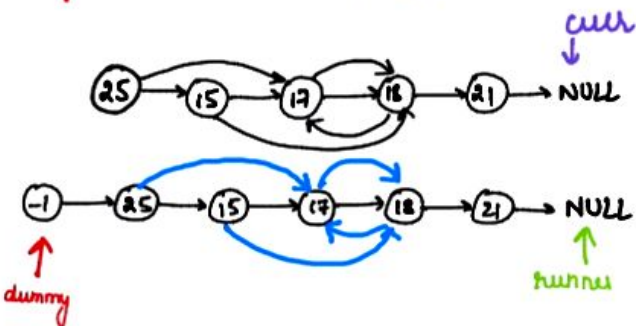
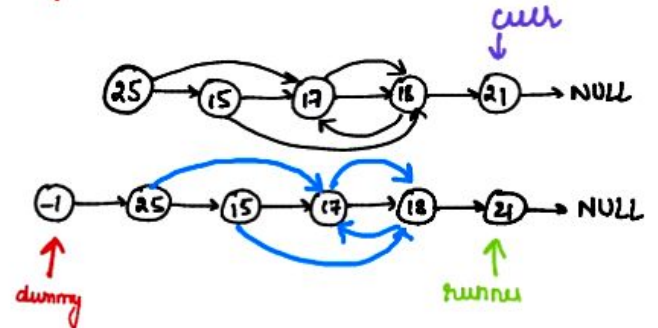
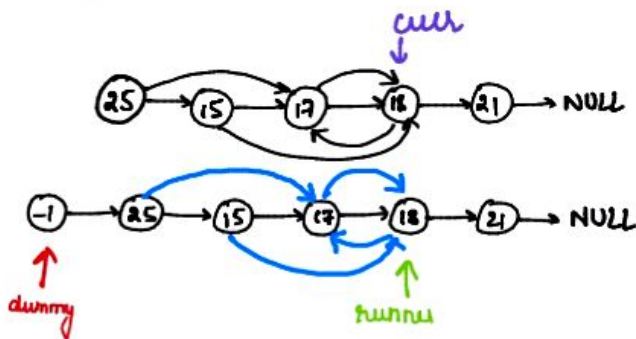
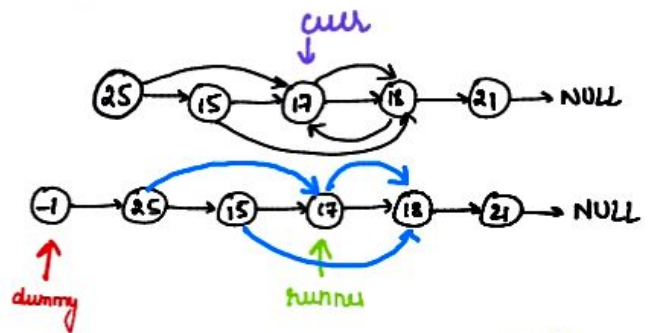
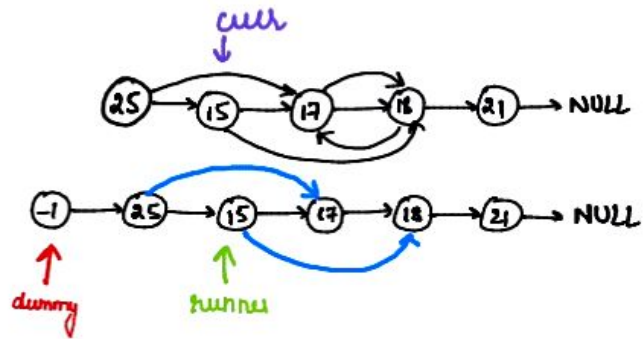
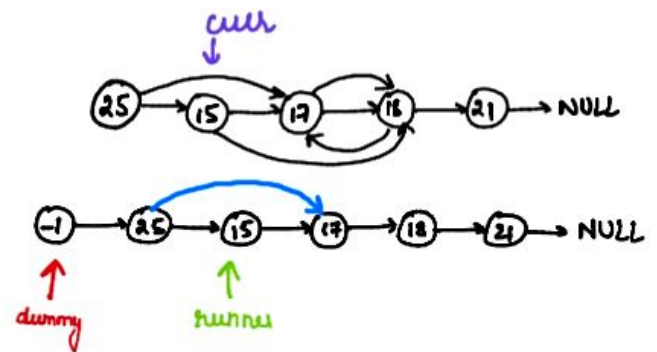
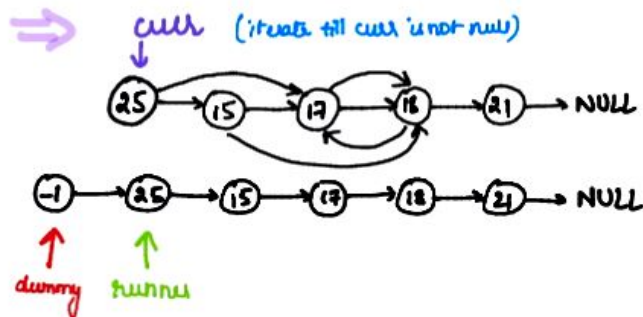
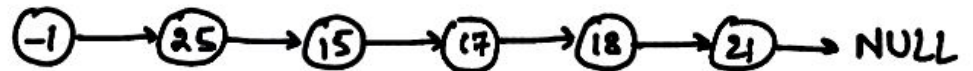


mp	
25	25
15	15
17	17
18	18
21	21
NULL	NULL
Old	New

→ In 1st iteration create the list without random pointer & also maintain hashmap for mapping node pointed by random pointer

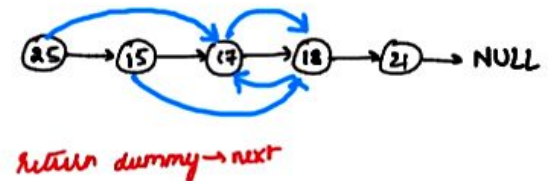
→ In 2nd iteration use **map** to link node pointed by random pointer

After 1st iteration →



→

→



return dummy → next



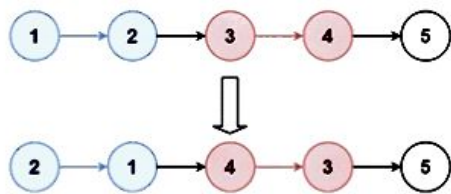
Code →

$T_c \rightarrow O(n)$

$Sc \rightarrow O(n)$

```
1  class Solution {
2  public:
3      Node* copyRandomList(Node* head) {
4
5          unordered_map<Node*, Node*> mp;
6          Node *dummy = new Node(100001);
7          Node *runner = dummy, *curr = head;
8
9          // initial iteration
10         while(curr != NULL){
11             Node *newNode = new Node(curr->val);
12             runner->next = newNode;
13             mp[curr] = newNode;
14             curr = curr->next;
15             runner = runner->next;
16         }
17
18         // setting starting points in both lists
19         curr = head;
20         runner = dummy->next;
21
22         // setting the random pointers
23         while(curr != NULL){
24             if(curr->random != NULL)
25                 runner->random = mp[curr->random];
26             runner = runner->next;
27             curr = curr->next;
28         }
29
30         return dummy->next;
31     }
32 };
```

# (1b) Reverse Nodes in K-Group



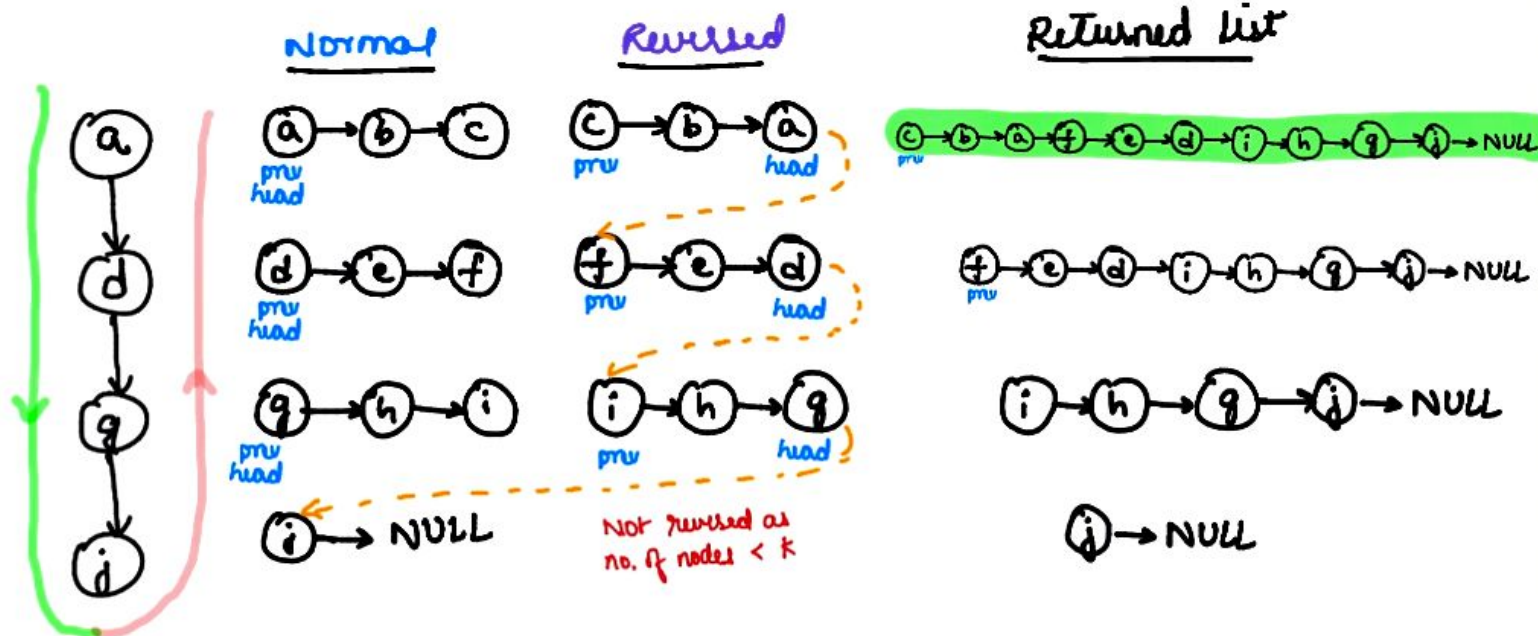
$K=2$

Given a linkedlist & K, return a list with reversed nodes by K-groups.

Ex.  $a \rightarrow b \rightarrow c \rightarrow d \rightarrow e \rightarrow f \rightarrow g \rightarrow h \rightarrow i \rightarrow j \rightarrow \text{NULL}$

$k=3$

Res  $\Rightarrow c \rightarrow b \rightarrow a \rightarrow f \rightarrow e \rightarrow d \rightarrow i \rightarrow h \rightarrow g \rightarrow j \rightarrow \text{NULL}$



\* Consider the case of  $f \rightarrow e \rightarrow d$ , then  $tempList = i \rightarrow h \rightarrow g$   
 linking would happen as  $head \rightarrow next = tempList$  & this list  
 would become tempList to  $c \rightarrow b \rightarrow a$

code →

$T_c \rightarrow O(n)$   
 $S_c \rightarrow O(1)$

```
1  class Solution {
2  public:
3      ListNode* reverseList(ListNode* head)
4      {
5          ListNode *prev = NULL, *curr = head, *temp;
6          while (curr!=NULL)
7          {
8              temp = curr->next;
9              curr->next = prev;
10             prev = curr;
11             curr = temp;
12         }
13         return prev;
14     }
15
16     ListNode* reverseInGroups(ListNode* head, int k)
17     {
18         ListNode *curr = head;
19         int currlen = 1;
20         if(head == NULL) return head;
21         while(curr->next!=NULL && currlen<k ){
22             curr=curr->next;
23             currlen+=1;
24         }
25         if(currlen<k) return head;
26         ListNode *tempNode = curr->next;
27         curr->next = NULL;
28
29         // start linking
30         ListNode *templist = reverseInGroups(tempNode,k);
31         ListNode *prev = reverseList(head);
32         head->next = templist;
33         return prev;
34     }
35
36     ListNode* reverseKGroup(ListNode* head, int k) {
37         return reverseInGroups(head,k);
38     }
39 };
```

# ⑦ Design Linked list → Implementation of Doubly Linked list

code →

```
class Node{
public:
    int val;
    Node* prev;
    Node* next;
    Node(int val){
        this->val=val;
        prev = nullptr;
        next = nullptr;
    }
};
class MyLinkedList {
public:
    Node *head;
    Node *tail;
    MyLinkedList(){
        head = nullptr;
        tail = nullptr;
    }

    int get(int index){
        if(head == NULL) return -1;
        Node *temp = head;
        int count = 0;
        while(temp!=NULL){
            temp=temp->next;
            count++;
        }
        if(index>=count) return -1;
        temp = head;
        while(temp != NULL && index>0){
            temp=temp->next;
            index--;
        }
        return temp->val;
    }

    void addAtHead(int val){
        Node *newNode = new Node(val);
        if(head == NULL){
            head = newNode;
            tail = newNode;
        } else {
            newNode->next = head;
            head->prev = newNode;
            head = newNode;
        }
    }

    void addAtTail(int val){
        Node *temp = head;
        if(head == NULL){
            Node *newNode = new Node(val);
            head = newNode;
            tail = newNode;
            return;
        }
        while(temp->next!=NULL){
            temp = temp->next;
        }
        Node *newNode = new Node(val);
        temp->next = newNode;
        newNode->prev = temp;
        tail = newNode;
    }
};
```

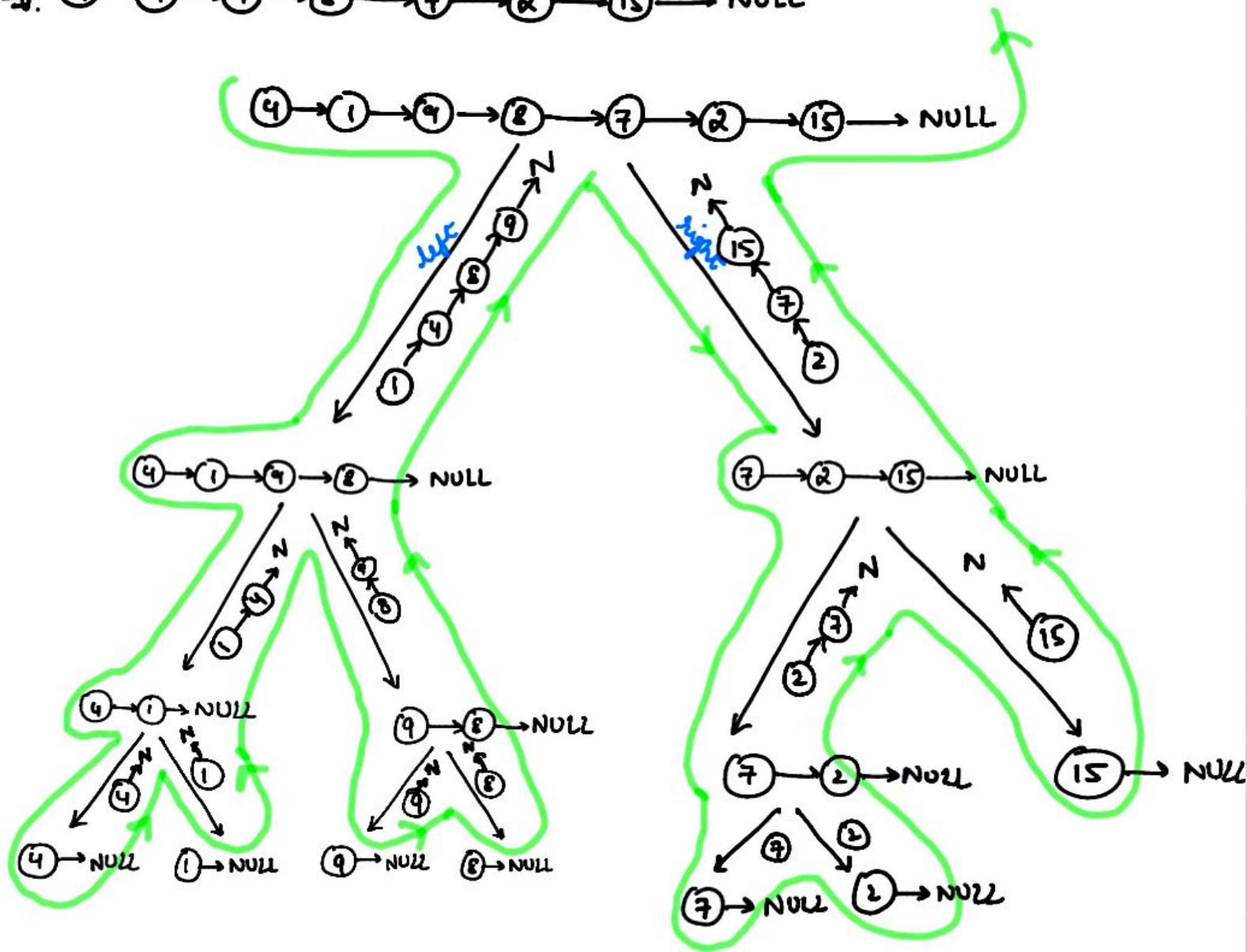
```
void addAtIndex(int index, int val){
    Node *temp = head;
    int count = 0;
    while(temp != NULL){
        temp = temp->next;
        count++;
    }
    if(index>count) return ;
    if(index==0){
        addAtHead(val);
        return;
    } else if(count == index){
        addAtTail(val);
        return;
    } else {
        temp = head;
        while(temp != NULL && index>0){
            temp = temp->next;
            index--;
        }
        Node* newNode = new Node(val);
        Node* temp2 = temp->prev;
        temp->prev->next = newNode;
        temp->prev = newNode;
        newNode->prev = temp2;
        newNode->next = temp;
    }
}

void deleteAtIndex(int index) {
    Node* temp = head;
    int count = 0;
    while(temp != NULL){
        temp=temp->next;
        count++;
    }
    if(index>=count) return;
    if(count==1 && index==0){
        head = NULL;
        return;
    } else if(count-1 == index){
        tail = tail->prev;
        tail->next = NULL;
        return;
    } else {
        if(index==0){
            head->next->prev = NULL;
            head = head->next;
            return;
        }
        temp=head;
        while(temp!=NULL && index>0){
            temp = temp->next;
            index--;
        }
        Node* temp2 = temp->next;
        temp->prev->next = temp2;
        temp->next->prev = temp->prev;
    }
}
```



18) Sort List → By following Merge Sort.

Ex. 4 → 1 → 9 → 8 → 7 → 2 → 15 → NULL



In the last step while returning from both branches we have,

left = 1 → 4 → 8 → 9 → NULL & right = 2 → 7 → 15 → NULL

so create dummy node & merge, i.e. -1 → 1 → 2 → 4 → 7 → 8 → 9 → 15 → NULL

return dummy → next, 1 → 2 → 4 → 7 → 8 → 9 → 15 → NULL

\* The same happens at every intermediate merge.

code →

$T_c \rightarrow O(m+n)$

$S_c \rightarrow O(n)$

```
1 class Solution {
2 public:
3     ListNode* merge(ListNode* l1, ListNode* l2) {
4         ListNode *dummy = new ListNode(-1);
5         ListNode *curr = dummy;
6         while(l1 && l2){
7             if(l1->val < l2->val){
8                 curr->next = l1;
9                 l1 = l1->next;
10            } else {
11                curr->next = l2;
12                l2 = l2->next;
13            }
14            curr = curr->next;
15        }
16        if(l1) curr->next = l1;
17        if(l2) curr->next = l2;
18
19        return dummy->next;
20    }
21
22    ListNode* sortList(ListNode* head) {
23        if(!head || !head->next) return head;
24
25        ListNode *slow = head;
26        ListNode *fast = head->next;
27        while(fast && fast->next) {
28            slow = slow->next;
29            fast = fast->next->next;
30        }
31        // dividing the lists into 2 parts
32        fast = slow->next;
33        slow->next = NULL;
34
35        // sort & merge
36        head = sortList(head);
37        fast = sortList(fast);
38        return merge(head, fast);
39    }
40 };
```

# Recursion & Backtracking

- Karun Karthik

## Contents →

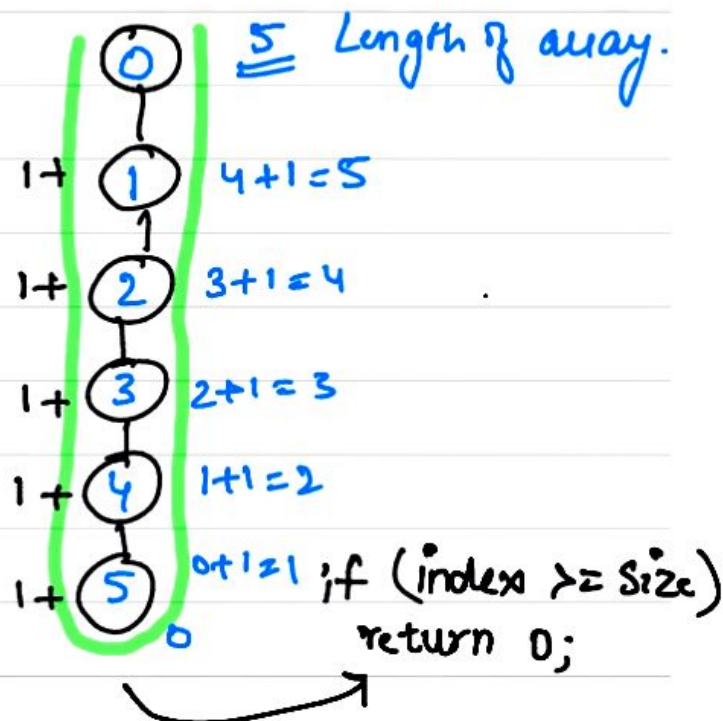
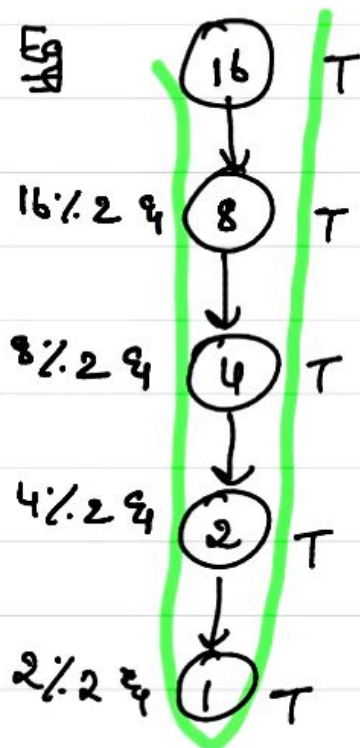
- |   |                             |
|---|-----------------------------|
| ① Power of two                          | ⑪ Subsets <u>II</u>         |
| ② Power of three                        | ⑫ Combination sum <u>II</u> |
| ③ Power of four                         | ⑬ N-Queens <u>I</u>         |
| ④ Subsets                               |                             |
| ⑤ Combination sum                       |                             |
| ⑥ Rat in a maze                         |                             |
| ⑦ N-Queens                              |                             |
| ⑧ Sudoku solver                         |                             |
| ⑨ Knight's tour problem                 |                             |
| ⑩ Letter combination of a phone number. |                             |

⑦ Length of an array

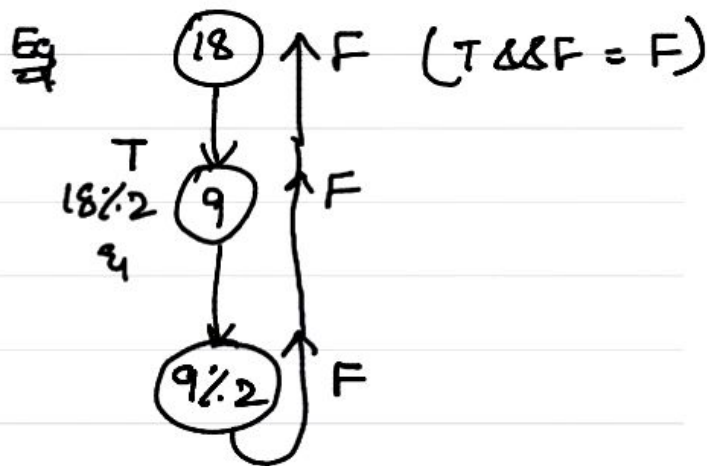
[20, 10, 40, 50, 30]  
 0 1 2 3 4

$TC = O(n)$

$SC = O(n)$

① Power of 2  $\rightarrow 2^k = 2^0 \cdot 2^1 \cdot 2^2 \dots 2^n$ 

if 1 then  
 return true.



$TC = O(\log_2 n)$



## Power of 2

C++ ▾

```
class Solution {
public:
    bool isPowerOfTwo(int n) {
        if(n==1) return true; //need to write it first else it might
        if(n<=0 || n%2!=0) return false;
        return isPowerOfTwo(n/2);
    }
};
```

## ② Power of 3

C++ ▾

```
class Solution {
public:
    bool isPowerOfThree(int n) {
        if(n==1) return true; //need to write it first else it might
        if(n<=0 || n%3!=0) return false;
        return isPowerOfThree(n/3);
    }
};
```

## ③ Power of 4

C++ ▾

```
class Solution {
public:
    bool isPowerOfFour(int n) {
        if(n==1) return true; //need to write it first else it might
        if(n<=0 || n%4!=0) return false;
        return isPowerOfFour(n/4);
    }
};
```

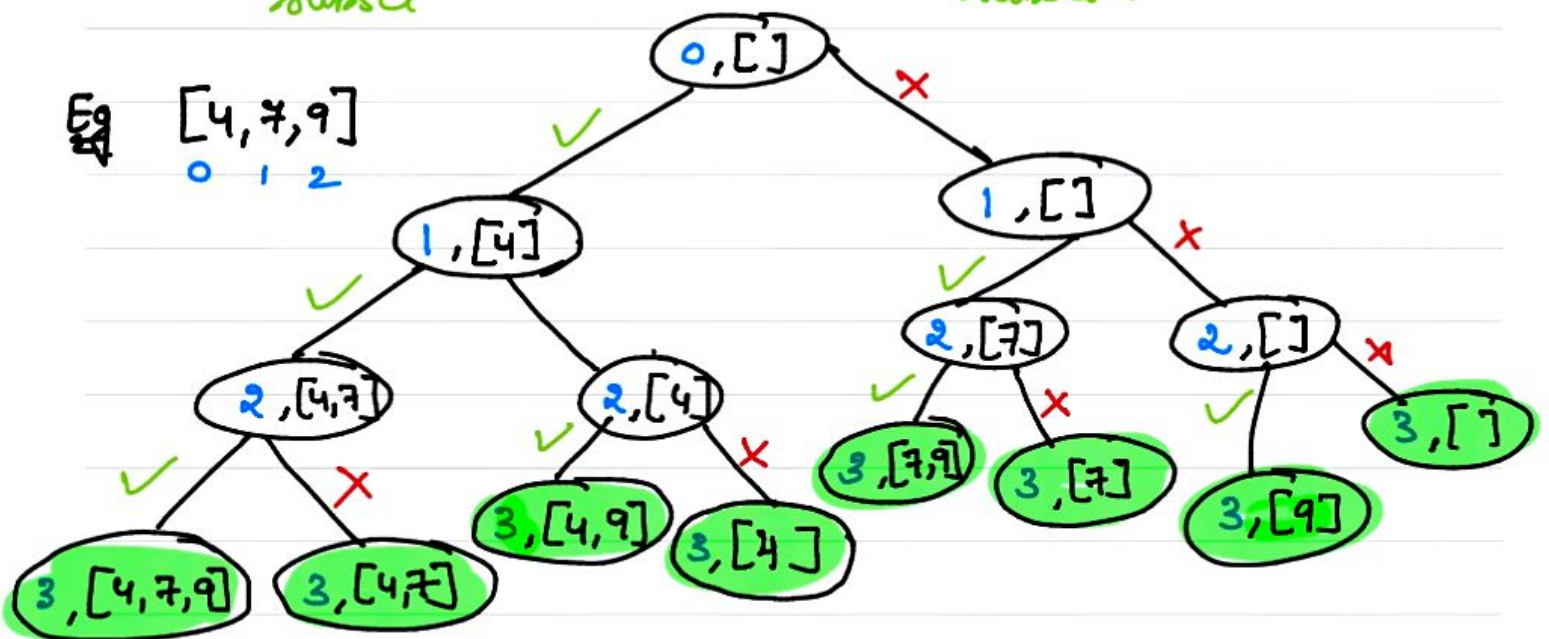
## D2 Subsets

- ④ Given an integer array nums, generate all the subsets. (Subsequences)  
If size = n then no. of subsets =  $2^n$ .

Eg nums = [1, 2, 3]

Sol = [[ ], [1], [1, 2], [1, 3], [1, 2, 3], [2], [2, 3], [3]]

For every, <sup>element</sup>  
can be a part of subset      cannot be a part of subset.



⇒ [ [4, 7, 9], [4, 7], [4, 9], [4], [7, 9], [7], [9], [ ] ].

\* Once index is greater than or equal to size then store in result

Tc =  $O(2^n)$  → as there are 2 possibilities at every element.  
Sc =  $O(2^n)$

## Code

```
class Solution {
public:
    void generateAllSubsets(vector<int>&nums, int currentIndex, vector<int>&res, vector<vector<int>> &powerSet){
        // base condition
        if(currentIndex >= nums.size()){
            powerSet.push_back(res);
            return;
        }
        int currentVal = nums[currentIndex];
        res.push_back(currentVal);
        generateAllSubsets(nums, currentIndex+1, res, powerSet);

        // remove the currentVal (not considering)
        res.pop_back();
        generateAllSubsets(nums, currentIndex+1, res, powerSet);
    }

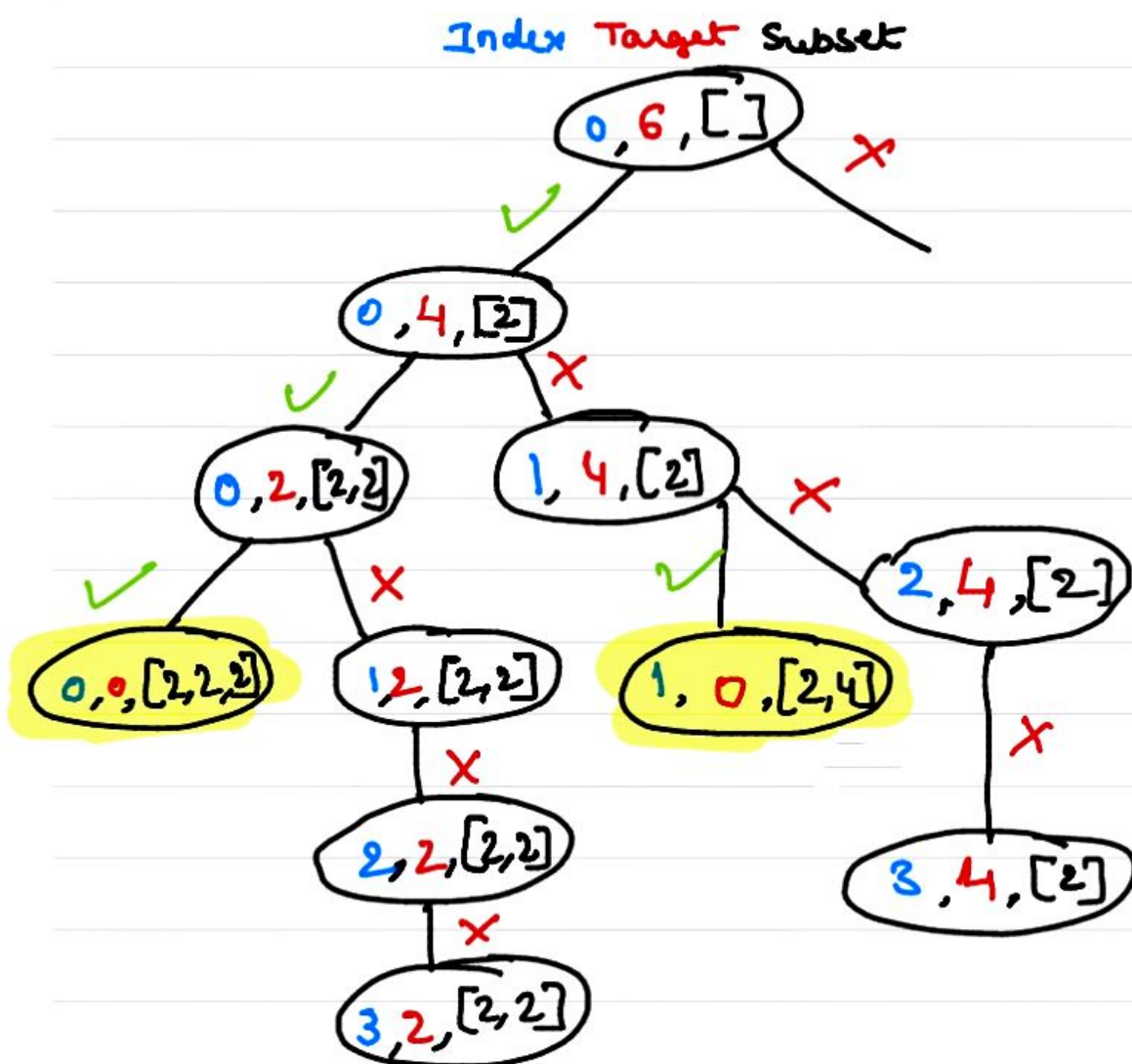
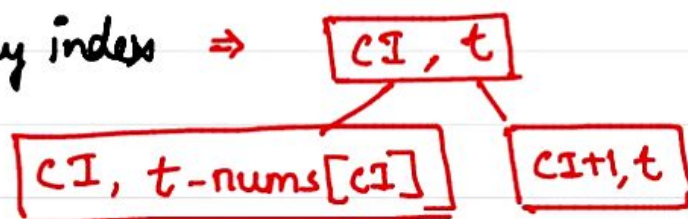
    vector<vector<int>> subsets(vector<int>& nums) {
        vector<vector<int>> powerSet;
        vector<int> res;
        generateAllSubsets(nums, 0, res, powerSet);
        return powerSet;
    }
};
```

⑤ Combination sum :-  $\text{nums} = [2, 3, 5]$   $\text{target} = 8$   
0 1 2

sol.  $[[2, 2, 2, 2], [2, 3, 3], [3, 5]]$

Eg  $[2, 4, 5]$   
0, 1, 2  
 $\text{target} = 6$

For every index  $\Rightarrow$



$\Rightarrow [[2, 2, 2], [2, 4]]$

\* Store the result when target sum = 0



Code →

```
class Solution {
public:
    void totalWays(vector<int>&candidates, int target, int curr, vector<vector<int>>&res, vector<int>&aux ){
        if(curr==candidates.size()){
            if(target==0){
                res.push_back(aux);
            }
            return;
        }
        // feasible only if curr value is less than the target
        if(candidates[curr]<=target){
            aux.push_back(candidates[curr]);
            totalWays(candidates, target-candidates[curr], curr,res,aux);
            aux.pop_back();
        }
        // back-tracking
        totalWays(candidates, target, curr+1,res,aux);
    }

    vector<vector<int>> combinationSum(vector<int>& candidates, int target) {
        vector<vector<int>> res;
        vector<int> aux;
        totalWays(candidates, target, 0, res, aux);
        return res;
    }
};
```

D3

⑥ Rat in a mazeGenerate all the ways to go from  $(0,0)$  to  $(n-1,n-1)$ 

\* At any cell we can move in D, L, R, U  
 $(x,c)$

$(x+1,c)$     $(x,c-1)$     $(x,c+1)$     $(x-1,c)$

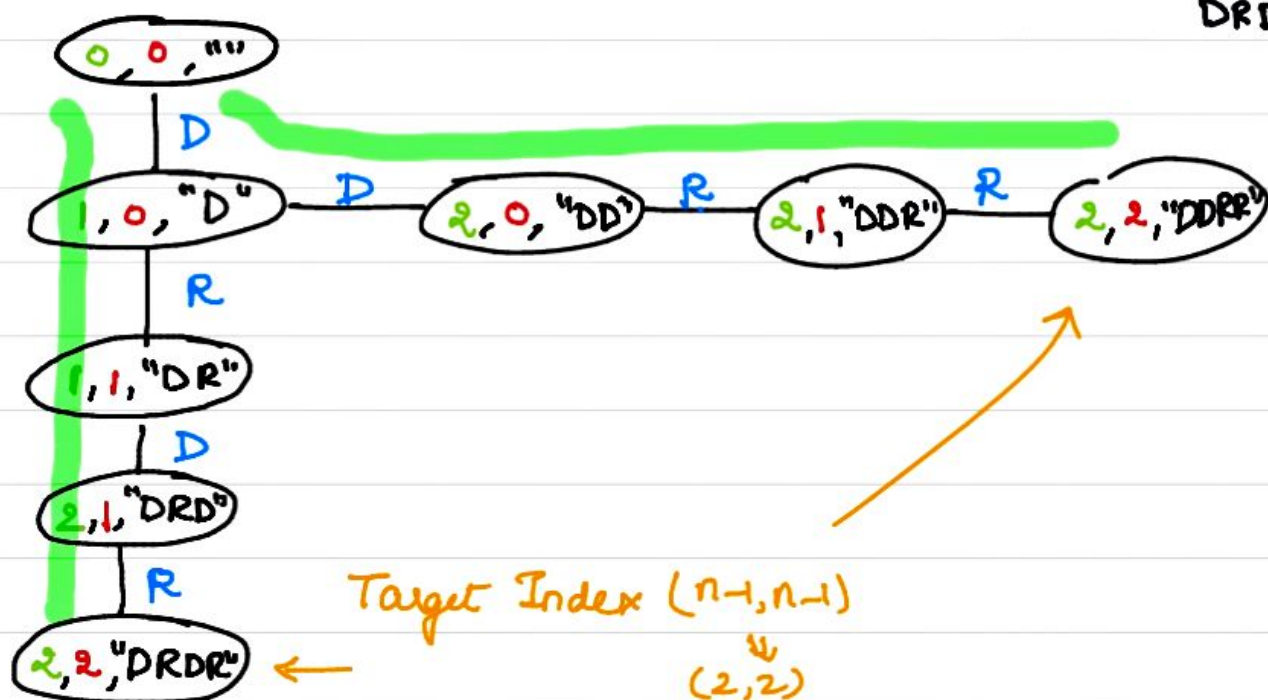
Q.  $n=3$

$\begin{bmatrix} 0 & 1 & 2 \\ [1, 0, 0], \\ [1, 1, 0], \\ [1, 1, 1] \end{bmatrix} \Rightarrow \text{DRDR, DDRR}$

$N=4$

$\begin{bmatrix} 0 & 0 & 0 & 0 \\ 1 & 1 & 0 & 1 \\ 1 & 1 & 0 & 0 \\ 0 & 1 & 1 & 1 \end{bmatrix}$

[DDRDRR, DRDDR]



\* Before making any call from cell change its state

\* while returning, **UNDO** the changes made (Backtracking.)

Code →

```
class Solution{
public:
    void allPaths(int row, int col, int n, vector<vector<int>>&m, string ans, vector<string>&res){

        if(row<0 || row>=n || col<0 || col>=n || m[row][col]==0){
            return;
        }

        if(row==n-1 && col==n-1){
            res.push_back(ans);
            return;
        }

        m[row][col]= 0;
        allPaths(row+1, col,n,m,ans+"D",res);
        allPaths(row, col-1,n,m,ans+"L",res);
        allPaths(row, col+1,n,m,ans+"R",res);
        allPaths(row-1, col,n,m,ans+"U",res);
        m[row][col] = 1;

        return;
    }

    vector<string> findPath(vector<vector<int>> &m, int n) {
        string ans = "";
        vector<string> res;
        allPaths(0,0,n,m,ans,res);
        sort(res.begin(), res.end());
        return res;
    }
};
```

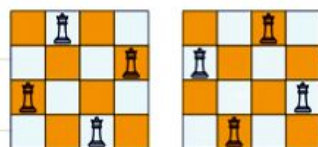
# N-queens

D4

return all configurations

⑦ If given  $n$ , then we should place  $n$ -queens in  $N \times N$  matrix, such that no 2 queens  $\rightarrow$  share same row, column, diagonal

Eg  $n=4$



Initially

	0	1	2	3
0	[., ., ., .]			
1	[., ., ., .]			
2	[., ., ., .]			
3	[., ., ., .]			

$X_R \rightarrow$  Bad Row

$X_C \rightarrow$  Bad column

$X_D \rightarrow$  Bad diagonal

$X_N \rightarrow$  Not possible

Step  $\rightarrow$  Start from (0,0)

①

	0	1	2	3
0	[Q, ., ., .]			
1	[., ., ., .]			
2	[., ., ., .]			
3	[., ., ., .]			

column pos  
0 1 2 3  
✓  
 $X_R$   $X_D$  ✓  $X_D$

②

	0	1	2	3
0	[Q, ., ., .]			
1	[., ., Q, .]			
2	[., ., ., .]			
3	[., ., ., .]			

column pos  
0 1 2 3  
✓  
 $X_R$   $X_D$  ✓  $X_D$   
 $X_R$   $X_D$   $X_C$   $X_D$   
 $\hookrightarrow$  this says (1,2) is a bad config & so is (0,0)  
 $\therefore$  Backtrack.

③

	0	1	2	3
0	[., ., Q, .]			
1	[., ., ., .]			
2	[., ., ., .]			
3	[., ., ., .]			

column pos  
0 1 2 3  
 $X_N$  ✓  
 $X_D$   $X_C$   $X_D$  ✓

④

	0	1	2	3
0	[., ., Q, .]			
1	[., ., ., Q]			
2	[., ., ., .]			
3	[., ., ., .]			

column pos  
0 1 2 3  
 $X_N$  ✓  
 $X_D$   $X_C$   $X_D$  ✓  
✓  $X_C$   $X_D$   $X_C$

⑤

	0	1	2	3
0	[., ., Q, .]			
1	[., ., ., Q]			
2	[Q, ., ., .]			
3	[., ., ., .]			

column pos  
0 1 2 3  
 $X_N$  ✓  
 $X_D$   $X_C$   $X_D$  ✓  
✓  $X_C$   $X_D$   $X_C$   
 $X_C$   $X_C$  ✓  $X_C$

⑥

	0	1	2	3
0	[., ., Q, .]			
1	[., ., ., Q]			
2	[Q, ., ., .]			
3	[., ., Q, .]			

column pos  
0 1 2 3  
 $X_N$  ✓  
 $X_D$   $X_C$   $X_D$  ✓  
✓  $X_C$   $X_D$   $X_C$   
 $X_C$   $X_C$  ✓  $X_C$

$\uparrow$  final result  $\therefore$  share & backtrack for other config.



code →

```
1  class Solution {
2  public:
3
4      bool valid_row(int curr_row, vector<vector<char>>&grid, int n){
5          for(int i = 0; i < n; i++){
6              if(grid[curr_row][i]=='Q')
7                  return false;
8          }
9          return true;
10     }
11
12     bool valid_col(int curr_col, vector<vector<char>>&grid, int n){
13         for(int i = 0; i < n; i++){
14             if(grid[i][curr_col]=='Q')
15                 return false;
16         }
17         return true;
18     }
19
20     bool valid_diagonal(vector<vector<char>>&grid, int curr_row, int curr_col, int n){
21         int i = curr_row;
22         int j = curr_col;
23         while(i>=0 && j>=0){           // Top-left diagonal
24             if(grid[i][j]=='Q')
25                 return false;
26             i--; j--;
27         }
28
29         i = curr_row;
30         j = curr_col;
31         while(i>=0 && j<n){           // Top-right diagonal
32             if(grid[i][j]=='Q')
33                 return false;
34             i--; j++;
35         }
36
37         i = curr_row;
38         j = curr_col;
39         while(i<n && j>=0){           // Bottom-left diagonal
40             if(grid[i][j]=='Q')
41                 return false;
42             i++; j--;
43         }
44
45         i = curr_row;
46         j = curr_col;
47         while(i<n && j<n){           // Bottom-right diagonal
48             if(grid[i][j]=='Q')
49                 return false;
50             i++; j++;
51         }
52
53         return true;
54     }
55 }
```

```

1  bool isValid(vector<vector<char>>&grid, int curr_row, int curr_col, int n){
2      return valid_row(curr_row, grid, n) && valid_col(curr_col, grid, n) && valid_diagonal(grid, curr_row, curr_col, n);
3  }
4
5  // Function to convert grid char to strings
6  vector<string> populate(vector<vector<char>>&grid, int n){
7      vector<string> result;
8      for(int i = 0; i<n; i++){
9          string temp = "";
10         for(int j=0; j<n; j++){
11             temp += grid[i][j];
12         }
13         result.push_back(temp);
14     }
15     return result;
16 }
17
18 void solve(vector<vector<char>>&grid, int curr_row, int n, vector<vector<string>>&ans){
19     if(curr_row==n){
20         vector<string> temp = populate(grid,n);
21         ans.push_back(temp);
22         return;
23     }
24     for(int curr_col=0; curr_col < n; curr_col++){
25         if(isValid(grid, curr_row, curr_col,n)){
26             grid[curr_row][curr_col] = 'Q';
27             solve(grid, curr_row+1, n, ans);
28             grid[curr_row][curr_col] = '.';
29         }
30     }
31 }
32
33 vector<vector<string>> solveNQueens(int n) {
34     vector<vector<string>> ans;
35     vector<vector<char>>grid(n, vector<char>(n,'.'));
36     solve(grid, 0, n, ans);
37     return ans;
38 }
39 };

```

### ⑬ N-Queens II

↳ need to find the total number of possibilities

⊛ everything is same as in N-Queens but return the no. of elements in the result.

⑧ A sudoku solution must satisfy all of the following rules:

- 1 Each of the digits 1-9 must occur exactly once in each row.
- 2 Each of the digits 1-9 must occur exactly once in each column.
- 3 Each of the digits 1-9 must occur exactly once in each of the 9 3x3 sub-boxes of the grid.

5P

5	3			7				
6			1	9	5			
	9	8					6	
8				6				3
4			8		3			1
7				2				6
	6					2	8	
			4	1	9			5
				8			7	9



5	3	4	6	7	8	9	1	2
6	7	2	1	9	5	3	4	8
1	9	8	3	4	2	5	6	7
8	5	9	7	6	1	4	2	3
4	2	6	8	5	3	7	9	1
7	1	3	9	2	4	8	5	6
9	6	1	5	3	7	2	8	4
2	8	7	4	1	9	6	3	5
3	4	5	2	8	6	1	7	9

### Algorithm

- ① Let  $(i, j)$  be an empty cell
- ② for  $i$  from 1 to 9:
  - if  $i$  is not in row, column, 3x3 sub-grid:
    - (a)  $grid(r, c) = i$
    - (b) recursively fill remaining empty cells.
    - (c) if recursion is successful:
      - return True
    - (d)  $grid(r, c) = '.'$  (backtracking)
- ③ return false

# Code

```
1  class Solution {
2  public:
3      bool valid_row(vector<vector<char>>&board, int currRow, int currVal){
4          for(int i=0; i<9; i++){
5              if(board[currRow][i]==currVal+'0'){
6                  return false;
7              }
8          }
9          return true;
10     }
11
12     bool valid_col(vector<vector<char>>&board, int currCol, int currVal){
13         for(int i=0; i<9; i++){
14             if(board[i][currCol]==currVal+'0'){
15                 return false;
16             }
17         }
18         return true;
19     }
20
21     bool valid_grid(vector<vector<char>>&board, int currRow, int currCol, int currVal){
22         int x = 3*(currRow/3);
23         int y = 3*(currCol/3);
24         for(int i=0; i<3; i++){
25             for(int j=0; j<3; j++){
26                 if(board[x+i][y+j]== currVal+'0'){
27                     return false;
28                 }
29             }
30         }
31         return true;
32     }
33
34     bool isValidCell(vector<vector<char>>&board, int currRow, int currCol, int currVal){
35         return valid_row(board, currRow, currVal) && valid_col(board, currCol, currVal) &&
36         valid_grid(board, currRow, currCol, currVal);
37     }
38
39 }
```



```
1
2     bool sudokuSolver(vector<vector<char>>&board, int currRow, int currCol){
3         if(currRow==9)
4             return true;
5
6         int nextRow = 0;
7         int nextCol = 0;
8
9         // find next possible row n column
10        if(currCol==8){
11            nextRow = currRow+1;
12            nextCol = 0;
13        } else {
14            nextRow = currRow;
15            nextCol = currCol+1;
16        }
17
18        // if not filled then call
19        if(board[currRow][currCol]!='.'){
20            return sudokuSolver(board, nextRow, nextCol);
21        }
22
23        // try all possibilities from 1 to 9 numbers
24        for(int currVal=1; currVal<10; currVal++){
25
26            // if valid then make the change
27            if(isValidCell(board, currRow, currCol, currVal)){
28                board[currRow][currCol] = '0'+currVal;
29
30                // if already solved then return true directly
31                if(sudokuSolver(board, nextRow, nextCol)==true)
32                    return true;
33
34                // backtracking
35                board[currRow][currCol] = '.';
36            }
37        }
38
39        return false;
40    }
41    void solveSudoku(vector<vector<char>>& board) {
42        sudokuSolver(board, 0, 0);
43    }
44    };
```

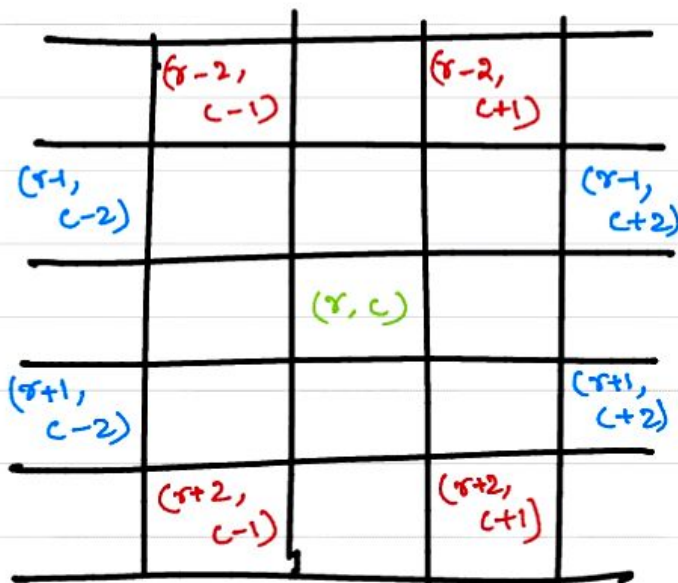
## D6 Knight's tour problem.

- ⑨ Given an  $n \times n$  board, print the order of each cell in which they are visited. ( $n \geq 8$ )

For  $n = 8$ , the result is

0	59	38	33	30	17	8	63
37	34	31	60	9	62	29	16
58	1	36	39	32	27	18	7
35	48	41	26	61	10	15	28
42	57	2	49	40	23	6	19
47	50	45	54	25	20	11	14
56	43	52	3	22	13	24	5
51	46	55	44	53	4	21	12

Sol For every cell  $(r, c)$  we have 8 possibilities,



- $(r-2, c-1)$
- $(r-2, c+1)$
- $(r+2, c-1)$
- $(r+2, c+1)$
- $(r-1, c-2)$
- $(r-1, c+2)$
- $(r+1, c-2)$
- $(r+1, c+2)$

- The rest is similar to rat-in-a-maze problem except that the value will be incremented by 1.

Code →

```
1  #include <bits/stdc++.h>
2  using namespace std;
3
4  void display(vector<vector<int>>&grid){
5      for(auto i: grid){
6          for(auto j:i){
7              cout<<j<<" ";
8          }
9          cout<<"\n";
10     }
11 }
12
13 void KnightTour(vector<vector<int>> &grid, int currRow, int currCol,
14                 int upcomingVal, int n){
15     if(upcomingVal==n*n){
16         display(grid);
17         cout<<"\n";
18         return;
19     }
20
21     if(currRow<0 || currRow>=n || currCol<0 || currCol>=n
22        || grid[currRow][currCol]!=0){
23         return;
24     }
25
26     grid[currRow][currCol] = upcomingVal;
27
28     KnightTour(grid, currRow-2, currCol-1, upcomingVal+1, n);
29     KnightTour(grid, currRow-2, currCol+1, upcomingVal+1, n);
30     KnightTour(grid, currRow+2, currCol-1, upcomingVal+1, n);
31     KnightTour(grid, currRow+2, currCol+1, upcomingVal+1, n);
32     KnightTour(grid, currRow-1, currCol-2, upcomingVal+1, n);
33     KnightTour(grid, currRow-1, currCol+2, upcomingVal+1, n);
34     KnightTour(grid, currRow+1, currCol-2, upcomingVal+1, n);
35     KnightTour(grid, currRow+1, currCol+2, upcomingVal+1, n);
36
37     grid[currRow][currCol] = 0;
38     return;
39 }
40
41 int main() {
42     int n;
43     cin>>n;
44     vector<vector<int>>grid(n,vector<int>(n,0));
45     KnightTour(grid, 0, 0, 1, n);
46     return 0;
47 }
48
```



# ⑩ Letter combination of a phone number

Eg digits = "23" → abc  
                  0 1       0 1 2  
                  0 1       0 1 2

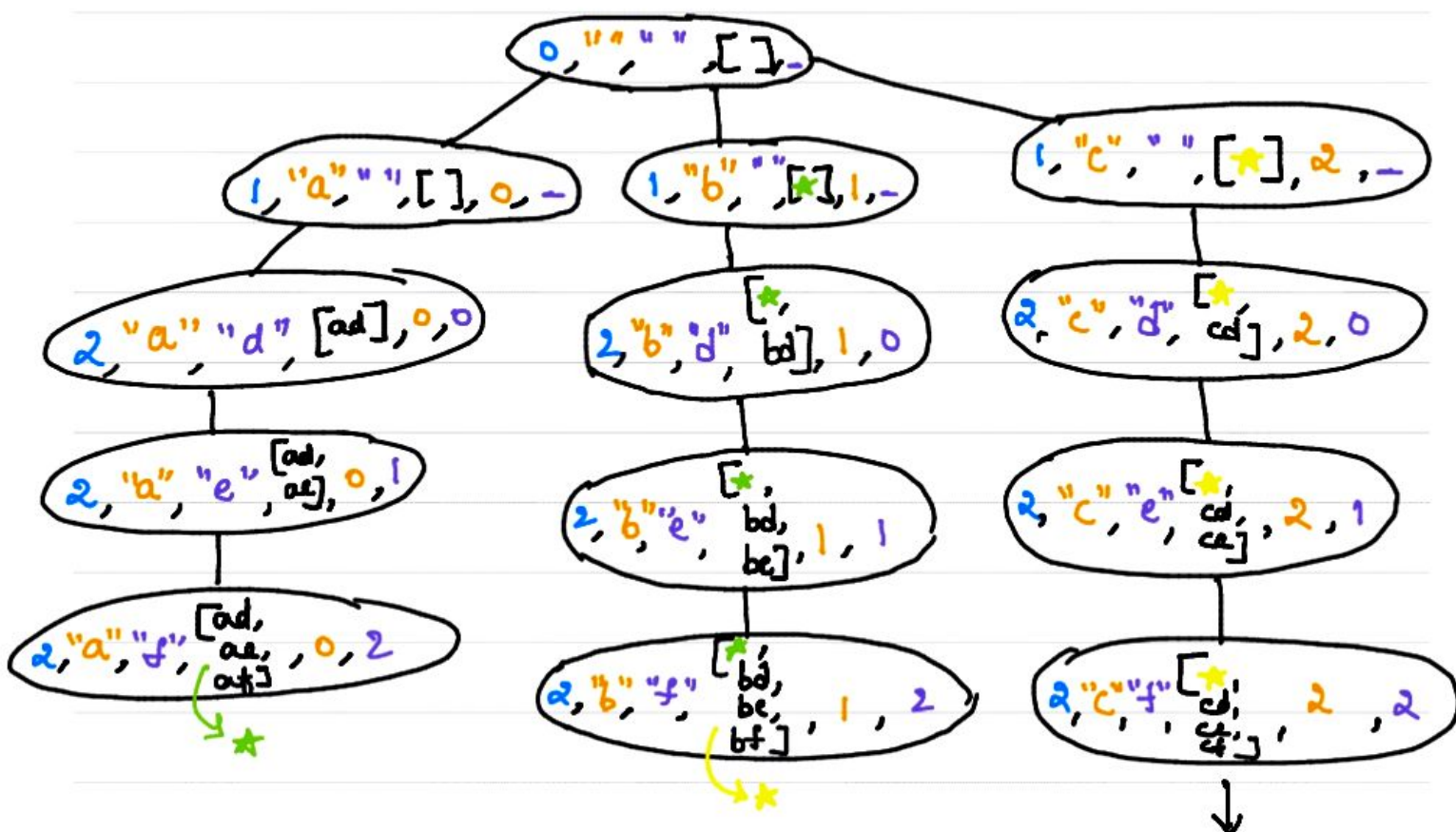
"2" → abc  
     0 1 2

"3" → def  
     0 1 2



\* Initially create a map for numbers & their alphabets

\* Then for each index in a string find all possibilities



final results.

[ad, ae, af, bd, be, bf, cd, ce, cf]



code →

```
1  class Solution {
2  public:
3      void findAll( map<char,string> &mapper, string digits,
4                  vector<string> &ans, string &s, int currentIndex){
5
6          if(currentIndex>=digits.length()){
7              ans.push_back(s);
8              return;
9          }
10
11         char currNum = digits[currentIndex];
12         string alpha = mapper[currNum];
13
14         for(int i=0; i<alpha.size(); i++){
15             s.push_back(alpha[i]);
16             findAll(mapper, digits, ans, s, currentIndex+1);
17             s.pop_back();
18         }
19         return;
20     }
21
22     vector<string> letterCombinations(string digits) {
23
24         map<char,string>mapper{
25             {'1', ""},
26             {'2', "abc"},
27             {'3', "def"},
28             {'4', "ghi"},
29             {'5', "jkl"},
30             {'6', "mno"},
31             {'7', "pqrs"},
32             {'8', "tuv"},
33             {'9', "wxyz"},
34         };
35         string s = "";
36         vector<string> ans;
37
38         // edge case
39         if(digits.size()==0){
40             return ans;
41         }
42         // else generate all possibilities
43         findAll(mapper, digits, ans, s, 0);
44         return ans;
45     }
46 }
47 };
```

⑮ Subsets II → same as subsets but no duplicates.

① using set<int>

Code →

```
1  class Solution {
2  public:
3      void allsubs(vector<int>& nums, int curr,
4      vector<int>& ds, set<vector<int>>& ans)
5      {
6          if(curr >= nums.size()){
7              ans.insert(ds);
8              return;
9          }
10         int currval = nums[curr];
11         ds.push_back(currval);
12         allsubs(nums, curr+1, ds, ans);
13
14         // removing currentVal (not considering)
15         ds.pop_back();
16         allsubs(nums, curr+1, ds, ans);
17     }
18
19     vector<vector<int>> subsetsWithDup(vector<int>& nums) {
20         set<vector<int>> ans;
21         vector<int> vec;
22         sort(nums.begin(), nums.end());
23         allsubs(nums, 0, vec, ans);
24         vector<vector<int>> res{ans.begin(), ans.end()};
25         return res;
26     }
27 };
```

## ② without using sets

code →

```
1  class Solution {
2  public:
3      void allsubs(vector<int> &nums, int curr, vector<int> &ds,
4                  vector<vector<int>>& res){
5          res.push_back(ds); // storing initial answers
6          for(int i=curr; i<nums.size(); i++){
7              if(i>curr && nums[i]==nums[i-1]) continue; // avoiding duplicates
8              ds.push_back(nums[i]);
9              allsubs(nums, i+1, ds, res);
10             ds.pop_back();
11         }
12         return;
13     }
14
15     vector<vector<int>> subsetsWithDup(vector<int>& nums) {
16         vector<vector<int>> res;
17         vector<int> ds;
18         sort(nums.begin(), nums.end());
19         allsubs(nums, 0, ds, res);
20         return res;
21     }
22 };
23
24
```

## ⑫ Combinational Sum - II

→ Same as combinational sum but no duplicates

Code →

```
1  class Solution {
2  public:
3      void findAll(vector<int>&candidates, int target, int idx,
4                  vector<vector<int>> &ans, vector<int> &ds){
5
6          if(target==0){
7              ans.push_back(ds);
8              return;
9          }
10
11         for(int i = idx; i<candidates.size(); i++){
12
13             // avoid duplicates
14             if(i>idx && candidates[i]==candidates[i-1]) continue;
15
16             if(candidates[i]<=target){
17                 ds.push_back(candidates[i]);
18                 findAll(candidates, target-candidates[i], i+1, ans, ds);
19                 ds.pop_back();
20             }
21         }
22     }
23
24     vector<vector<int>> combinationSum2(vector<int>& candidates,
25                                       int target){
26         vector<vector<int>> ans;
27         sort(candidates.begin(), candidates.end());
28         vector<int> ds;
29         findAll(candidates, target, 0, ans, ds);
30         return ans;
31     }
32 };
```



### ⑬ N-Queens II

↳ need to find the total number of possibilities

⊛ everything is same as in N-Queens but return the no. of elements in the result.

# *Trees - Part 1*

---

- Karun Karthik

## *Contents*

0. Introduction
1. Max depth of Binary tree
2. Max depth of N-ary tree
3. Preorder of binary tree
4. Preorder of N-ary tree
5. Postorder of binary tree
6. Postorder of N-ary tree
7. Inorder of Binary tree
8. Merge two binary trees
9. Sum of root to leaf paths
10. Uni-valued Binary tree
11. Leaf similar trees
12. Binary tree paths
13. Sum of Left leaves
14. Path sum
15. Left view of Binary tree
16. Right view of Binary tree
17. Same tree
18. Invert Binary tree
19. Symmetric tree
20. Cousins of Binary tree

# Trees

why trees?

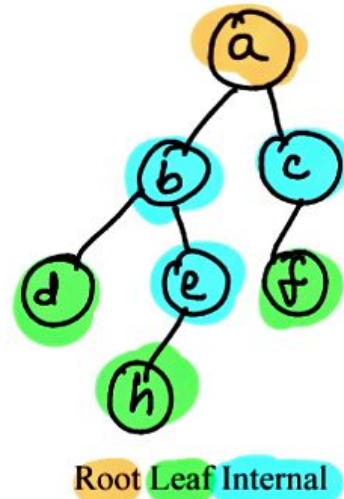
Tree - collection of tree-nodes

## ① Class Treenode

→ data  
→ list <Treenode> children

## ② Binary Tree → atmost 2 children (0,1,2)

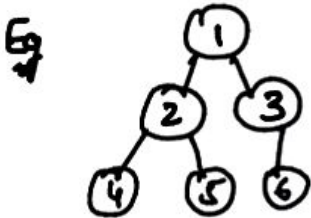
→ data  
→ leftchild  
→ rightchild



## ③ Types →

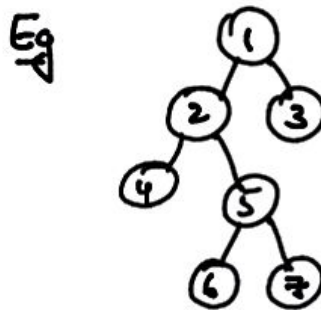
### ① Complete Binary Tree

→ all levels are completely filled except last one



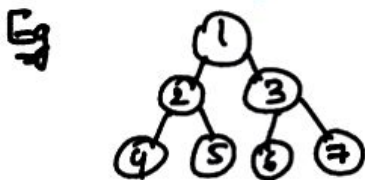
### ② Full Binary Tree

→ if every node has 0 or 2 children



### ③ Perfect Binary Tree

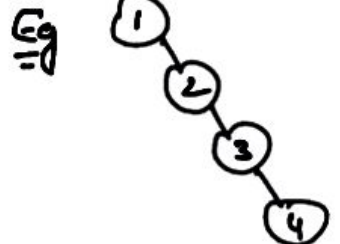
→ every internal node has exactly 2 children



### ④ Skewed Binary Tree

(\* used for finding complexity)

→ all nodes have either one or no child.



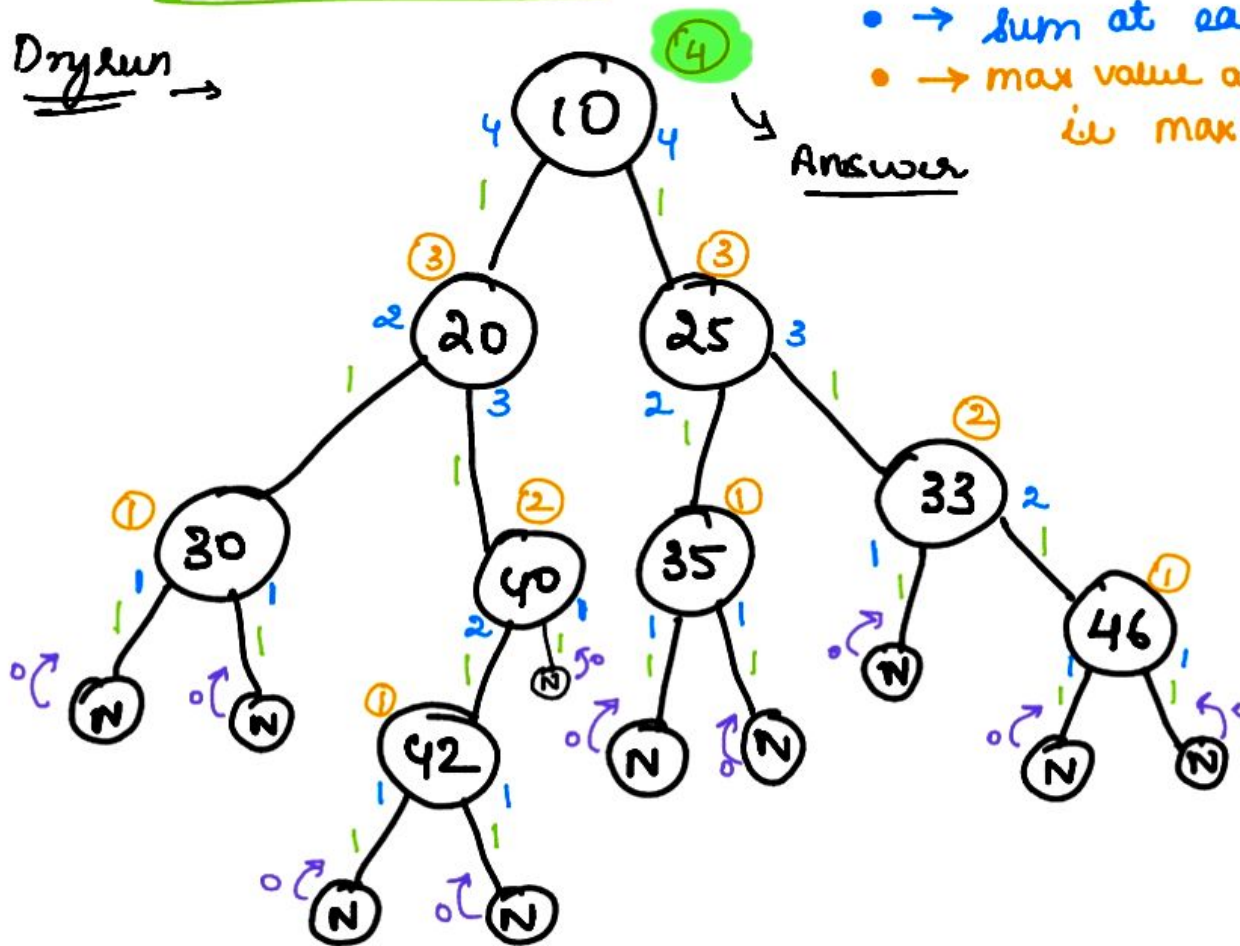
D1

① Depth of a binary tree (Max depth) • → 1 added while returning.

• → sum at each node.

• → max value at node is  $\max(\text{left}, \text{right})$ .

Dryrun →



if null  
then  $ht = 0$

• consider max  
at a node is  
either left or right

$T_c \rightarrow O(n)$

$SC \rightarrow O(1)$

$Aux \rightarrow O(h)$

$h \rightarrow \text{height}$

Code →

```
C++  
/**  
 * Definition for a binary tree node.  
 * struct TreeNode {  
 *     int val;  
 *     TreeNode *left;  
 *     TreeNode *right;  
 *     TreeNode() : val(0), left(nullptr), right(nullptr) {}  
 *     TreeNode(int x) : val(x), left(nullptr), right(nullptr) {}  
 *     TreeNode(int x, TreeNode *left, TreeNode *right) : val(x), left(left), right(right) {}  
 * };  
 */  
class Solution {  
public:  
    int maxDepth(TreeNode* root) {  
        if(root == NULL) return 0;  
  
        int lefth = 1 + maxDepth(root->left);  
        int righth = 1 + maxDepth(root->right);  
        return max(lefth, righth);  
    }  
};
```



## ② Maximum depth of n-ary tree

Idea is same as previous problem, only implementation changes

Code →

```
C++ v
/*
// Definition for a Node.
class Node {
public:
    int val;
    vector<Node*> children;

    Node() {}

    Node(int _val) {
        val = _val;
    }

    Node(int _val, vector<Node*> _children) {
        val = _val;
        children = _children;
    }
};
*/

class Solution {
public:
    int maxDepth(Node* root) {
        if(root==NULL) return 0;
        int ans=0;
        for(int i=0;i<root->children.size();i++)
        {
            int tempans = maxDepth(root->children[i]);
            ans = max(ans,tempans);
        }
        return ans+1;
    }
};
```

D2

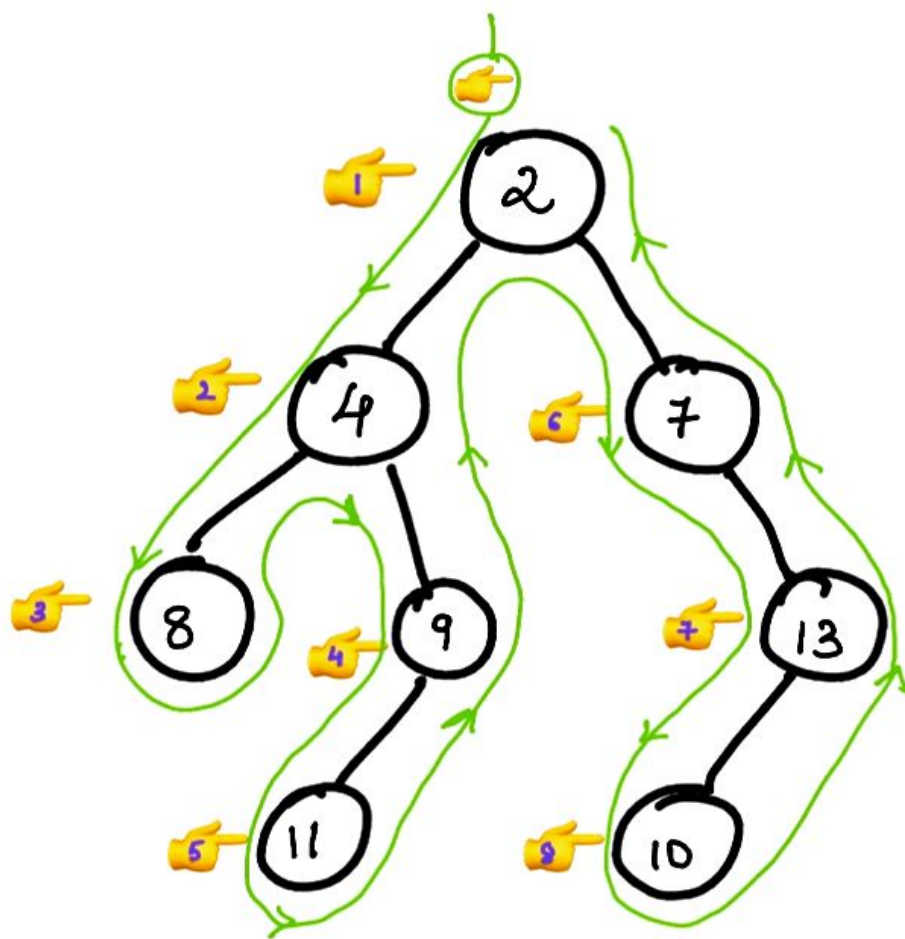
Traversals  $\rightarrow$  DFS  $\begin{cases} \rightarrow \text{Preorder} \\ \rightarrow \text{Inorder} \\ \rightarrow \text{Postorder} \end{cases}$   
 $\rightarrow$  BFS  $\rightarrow$  Level order

④ Preorder  $\rightarrow$

processing order

node  
left child  
right child

Eg



\* Point finger as shown  
and traverse the  
tree starting from Root

\* Order of visiting is the  
preorder traversal.

$[2, 4, 8, 9, 11, 6, 13, 10]$

Tc  $\rightarrow O(n)$

Sc  $\rightarrow O(h)$

Recursive Stack space  $\rightarrow O(h)$   $h \rightarrow$  height.

### ③ Pre-order traversal of Binary tree

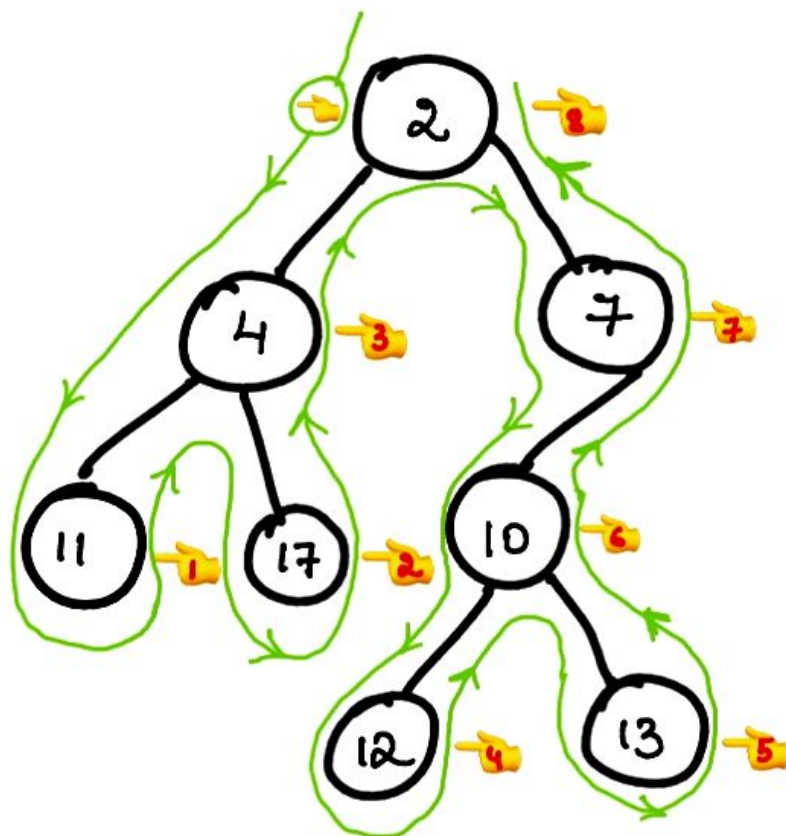
```
class Solution {
public:
    vector<int> preorderTraversal(TreeNode* root) {
        vector<int>ans;
        Preorder(root,ans);
        return ans;
    }
    void Preorder(TreeNode* root,vector<int>&ans)
    {
        if(root == NULL) return;
        ans.push_back(root->val);
        Preorder(root->left,ans);
        Preorder(root->right,ans);
        return;
    }
};
```

### ④ Pre-order traversal of n-ary tree

```
class Solution {
public:
    vector<int> preorder(Node* root) {
        vector<int>ans;
        Preorder(root,ans);
        return ans;
    }
    void Preorder(Node* root, vector<int>&ans)
    {
        if(root==NULL) return;
        ans.push_back(root->val);
        for(int i=0;i<root->children.size();i++)
        {
            Preorder(root->children[i],ans);
        }
        return;
    }
};
```

② Postorder → processing order → left child  
right child  
node

Eg



\* Point finger as shown  
and traverse the  
tree starting from Root

\* Order of visiting is the  
postorder traversal.

Tc →  $O(n)$

Sc →  $O(n)$

Recursive Stack space →  $O(h)$   $h \rightarrow$  height.

[11, 17, 4, 12, 13, 10, 7, 2]



## ⑤ Postorder traversal of Binary tree

```
class Solution {
public:
    vector<int> postorderTraversal(TreeNode* root) {
        vector<int>ans;
        Postorder(root,ans);
        return ans;
    }
    void Postorder(TreeNode* root,vector<int>&ans)
    {
        if(root == NULL) return;

        Postorder(root->left,ans);
        Postorder(root->right,ans);
        ans.push_back(root->val);
        return;
    }
};
```

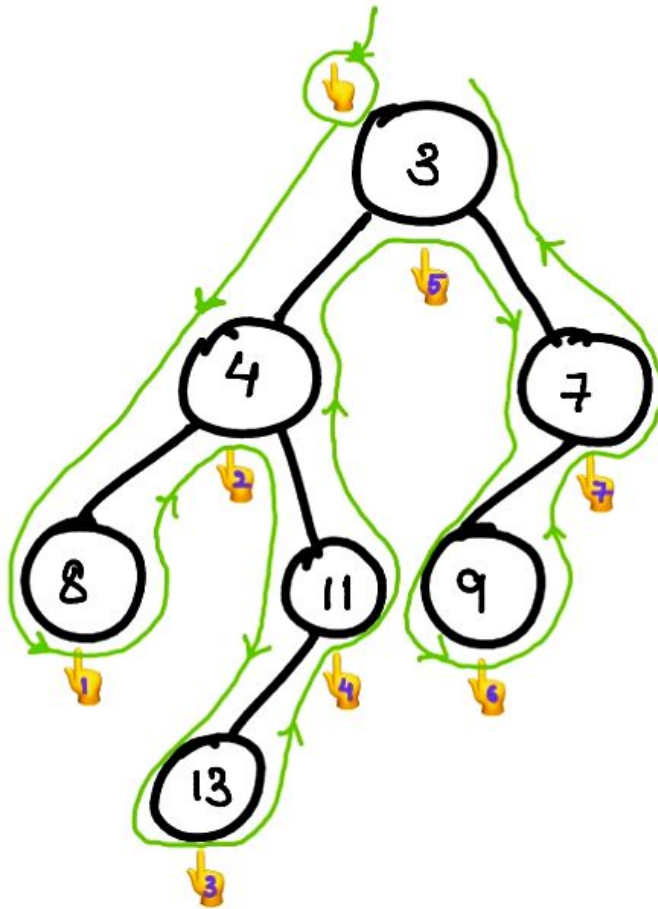
## ⑥ Postorder traversal of nary tree

```
class Solution {
public:
    vector<int> postorder(Node* root) {
        vector<int>ans;
        Postorder(root,ans);
        return ans;
    }
    void Postorder(Node* root, vector<int>&ans)
    {
        if(root == NULL) return;
        for(int i=0;i<root->children.size();i++)
        {
            Postorder(root->children[i],ans);
        }
        ans.push_back(root->val);
        return;
    }
};
```

### (c) Inorder →

processing order →  
left child  
node  
right child

Eg



\* Point finger as shown  
and traverse the  
tree starting from Root

\* Order of visiting is the  
Inorder traversal.

~~[8, 4, 13, 11, 3, 9, 7]~~

Tc →  $O(n)$

Sc →  $O(n)$

Recursive Stack space →  $O(h)$   $h \rightarrow$  height.

## ⑦ In-order traversal of Binary tree

```
class Solution {
public:
    vector<int> inorderTraversal(TreeNode* root) {
        vector<int>ans;
        Inorder(root,ans);
        return ans;
    }
    void Inorder(TreeNode* root, vector<int>&ans)
    {
        if(root==NULL) return;
        Inorder(root->left,ans);
        ans.push_back(root->val);
        Inorder(root->right,ans);
        return;
    }
};
```

## Inorder traversal of n-ary tree

Approach:

The inorder traversal of an N-ary tree is defined as **visiting all the children except the last then the root and finally the last child recursively.**

- Recursively visit the first child.
- Recursively visit the second child.
- .....
- Recursively visit the second last child.
- Print the data in the node.
- Recursively visit the last child.
- Repeat the above steps till all the nodes are visited.

```
void inorder(Node *node)
{
    if (node == NULL)
        return;

    // Total children count
    int total = node->length;

    // All the children except the last
    for (int i = 0; i < total - 1; i++)
        inorder(node->children[i]);

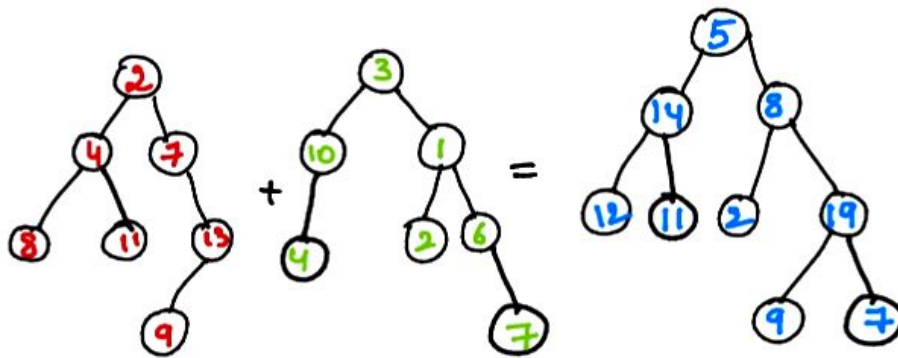
    // Print the current node's data
    cout<< node->data << " ";

    // Last child
    inorder(node->children[total - 1]);
}
```

### D3 ⑧ Merge two Binary trees →

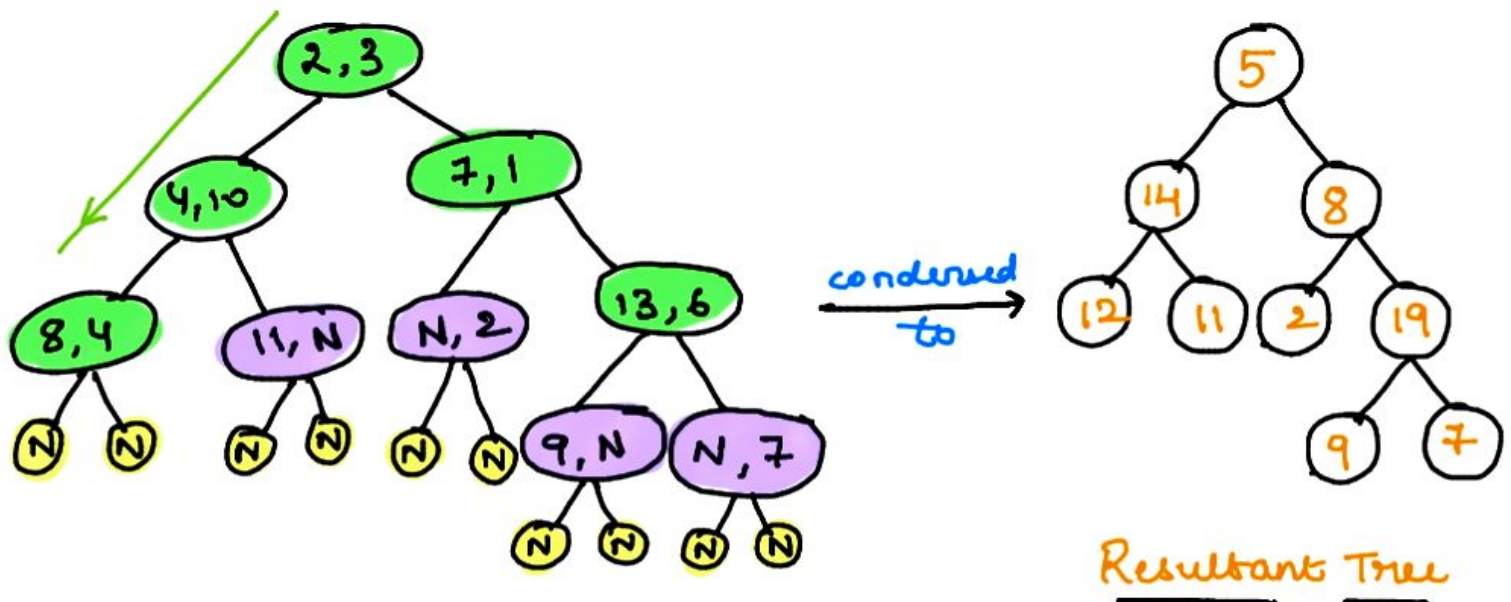
Given root nodes of 2 binary trees, return root of the sum tree.

Ex



we will perform preorder traversal on the binary tree because the node/root needs to be processed first.

The recursive tree structure would be like :



- NULL & NULL
- Node & NULL
- Node & Node

Tc →  $O(n+m)$

Sc →  $O(\max(n, m))$

Recursive stack →  $O(\max(h_1, h_2))$



code →

```
class Solution {
public:
    TreeNode* merge(TreeNode* root1, TreeNode* root2){

        if(root1==NULL && root2==NULL) return NULL;
        if(root1==NULL) return root2;
        if(root2==NULL) return root1;

        // Create new node to store sum
        TreeNode *newNode = new TreeNode(root1->val+root2->val);

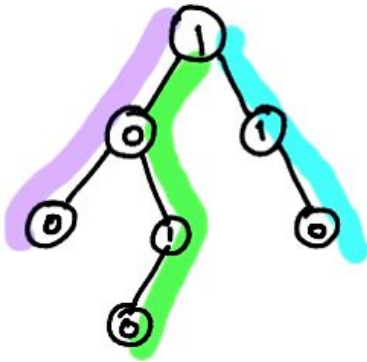
        // Recursively call the left sub-trees and right sub-trees
        newNode->left = merge(root1->left, root2->left);
        newNode->right = merge(root1->right, root2->right);

        // return the new node
        return newNode;
    }

    TreeNode* mergeTrees(TreeNode* root1, TreeNode* root2) {
        return merge(root1, root2);
    }
};
```

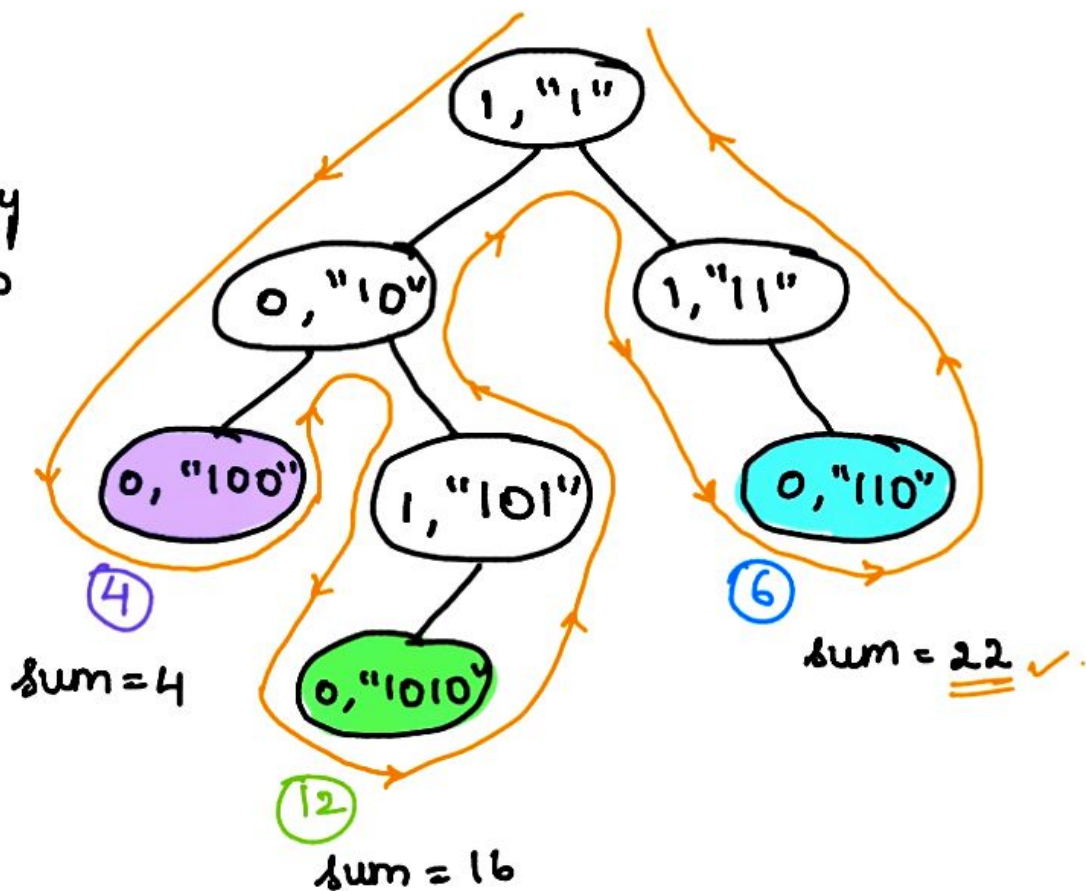
## ⑨ Sum of root to leaf paths →

Eg



$$\begin{aligned} &\rightarrow (100)_2 + (1010)_2 + (110)_2 \\ &\quad \text{4} + \text{12} + \text{6} \\ &= \underline{\underline{22}} \end{aligned}$$

Initially  
sum = 0



\* If root becomes null convert string to integer & add to sum.

Time →  $O(n)$

Space →  $O(n)$

Recursive stack →  $O(h)$

## Code

```
class Solution {
public:
    void rootToLeaf(TreeNode* root, string currentString, int* ans)
    {
        if(root->left== NULL && root->right==NULL)
        {
            currentString+=to_string(root->val);
            ans[0]+=stoi(currentString,0,2);
            return;
        }
        string curr=to_string(root->val);
        if(root->left!=NULL)
            rootToLeaf(root->left,currentString+curr,ans);
        if(root->right!=NULL)
            rootToLeaf(root->right,currentString+curr,ans);
    }
    int sumRootToLeaf(TreeNode* root) {
        int* ans=new int[1];
        ans[0]=0;
        rootToLeaf(root,"",ans);
        return ans[0];
    }
};
```

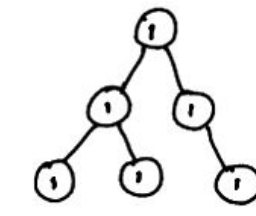
Note →

stoi() can take upto three parameters, the second parameter is for starting index and third parameter is for base of input number.

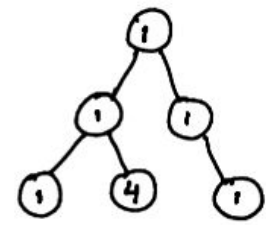
[to convert from binary to decimal we give it as 2]

(10) Univalued Binary Tree →

Ex

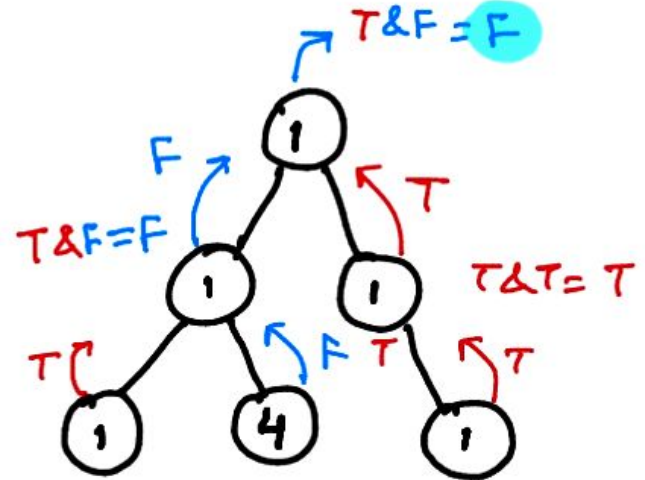
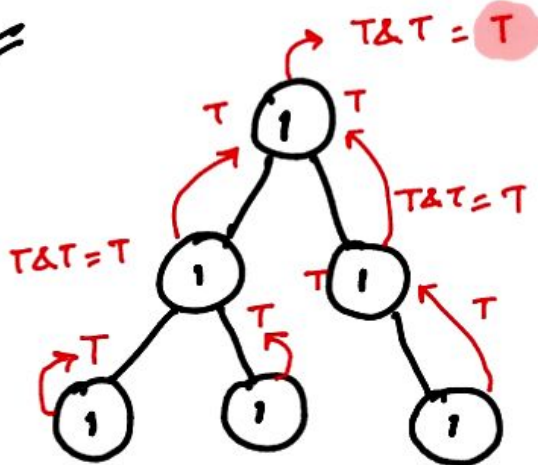


↳ returns true



↳ returns false

==



Code

```

class Solution {
public:
    bool isSame(TreeNode* root, int val){
        if(root==NULL) return true;
        if(root->val!=val) return false;

        bool left = isSame(root->left, val);
        bool right = isSame(root->right, val);

        return left && right;
    }

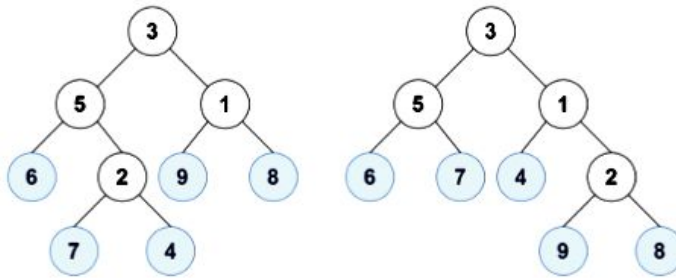
    bool isUnivalTree(TreeNode* root) {
        return isSame(root, root->val);
    }
};
  
```



# ① Leaf Similar trees

↳ return true if all leaves are in same order for both trees.

eg



$V_1 = 6, 7, 4, 9, 8 \Rightarrow V_1 = V_2$

$V_2 = 6, 7, 4, 9, 8$

↳ return true else false.

code →

```
class Solution {
public:
    void traversal(TreeNode* root, vector<int>&v){
        if(root==NULL)
            return;

        if(root->left==NULL && root->right==NULL)
            v.push_back(root->val);

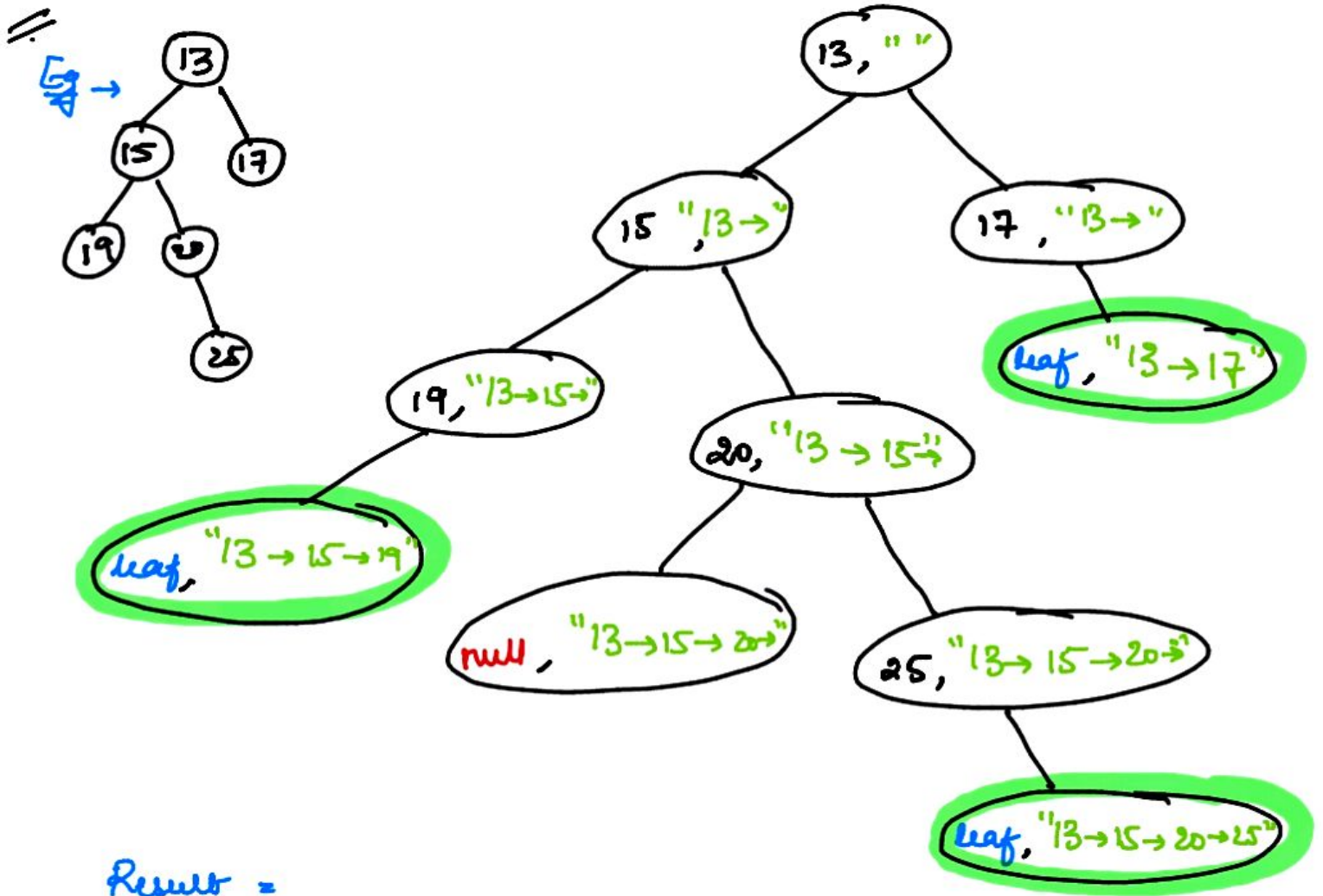
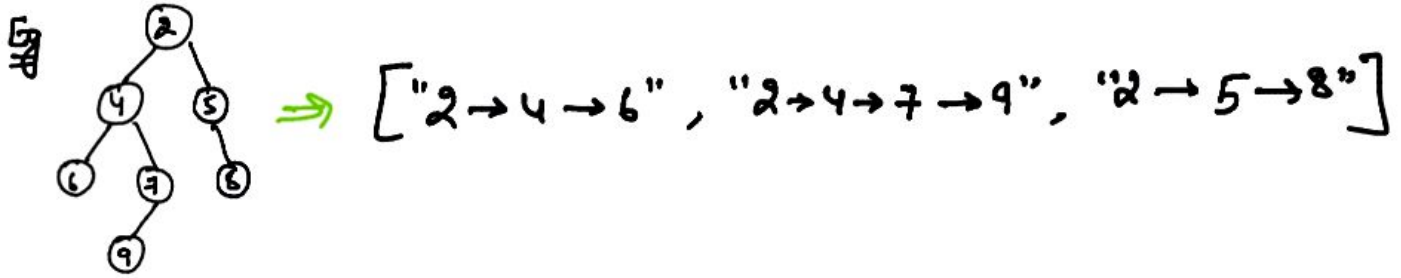
        if(root->left!=NULL)
            traversal(root->left, v);

        if(root->right!=NULL)
            traversal(root->right, v);
    }

    bool leafSimilar(TreeNode* root1, TreeNode* root2) {
        vector<int> a;
        vector<int> b;
        traversal(root1,a);
        traversal(root2,b);
        return a==b;
    }
};
```

## D5 (12) Binary tree paths

↳ given root print all the paths from root to leaf.



Time complexity =  $O(n)$

Space complexity =  $O(n) + O(h)$  → recursive stack.  
 ↳ Answer array

Code →

```
class Solution {
public:
    void pathFinder(TreeNode *root, vector<string> &res, string currPath){

        if(root==NULL) return;

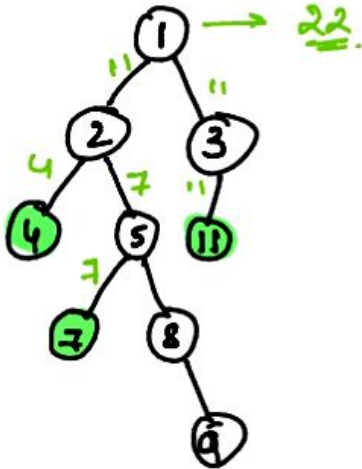
        // if leaf then add it's value to currentPath
        if(root->left == NULL && root->right==NULL){
            currPath += to_string(root->val);
            res.push_back(currPath);
            return;
        }

        // else add the node's value to path
        currPath += to_string(root->val)+"->";

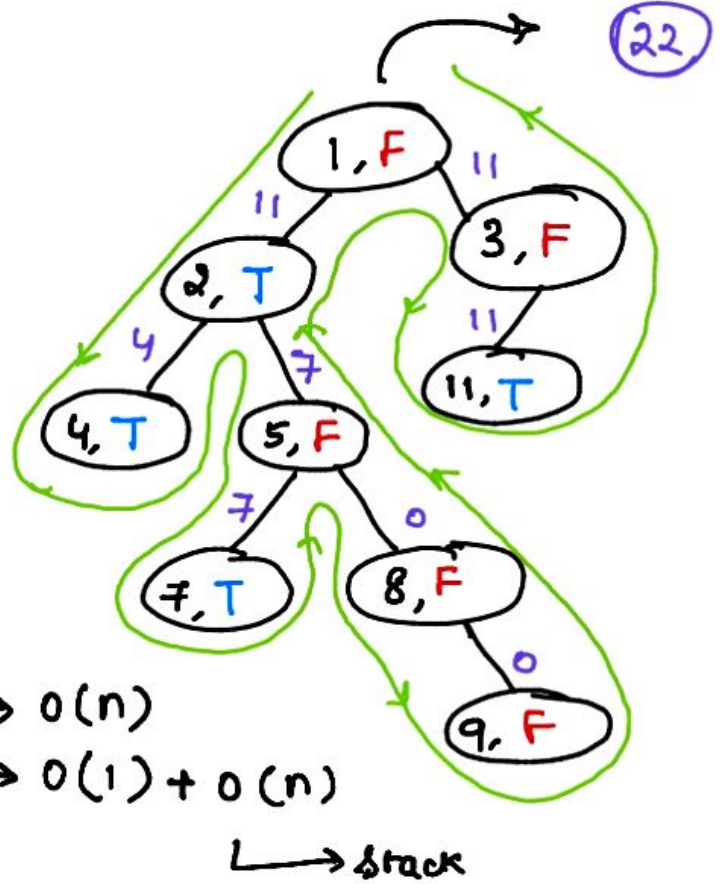
        if(root->left) pathFinder(root->left, res, currPath);
        if(root->right) pathFinder(root->right, res, currPath);
    }

    vector<string> binaryTreePaths(TreeNode* root) {
        vector<string> res;
        pathFinder(root, res, "");
        return res;
    }
};
```

⑬ sum of left leaves  $\rightarrow$



$$\begin{aligned} \text{Result} &= 4 + 7 + 11 \\ &= \underline{\underline{22}} \end{aligned}$$



Code →

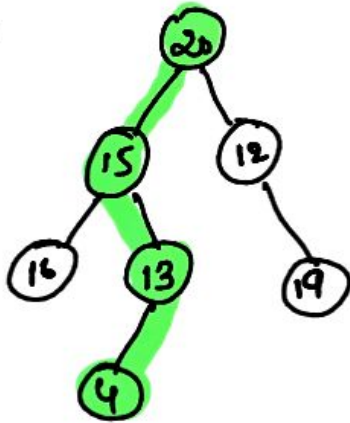
```
class Solution {
public:
    int leftLeafSum(TreeNode *root, bool leaf){
        if(root==NULL){
            return 0;
        }
        if(root->left==NULL && root->right==NULL && leaf){
            return root->val;
        }
        int ls = leftLeafSum(root->left, true);
        int rs = leftLeafSum(root->right, false);
        return ls+rs;
    }

    int sumOfLeftLeaves(TreeNode* root) {
        return leftLeafSum(root, false);
    }
};
```

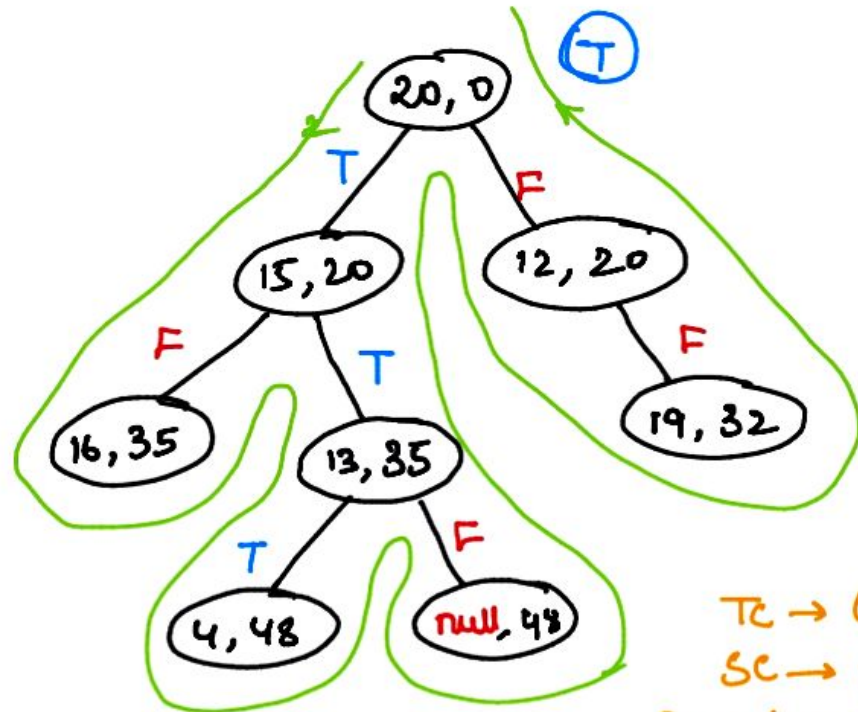


(14) Path sum → sum of all nodes from root to leaf is equal to target sum → then T else F.

eg



target = 52



TC →  $O(n)$   
 SC →  $O(1)$   
 Recursive →  $O(h)$   
 Stack

code

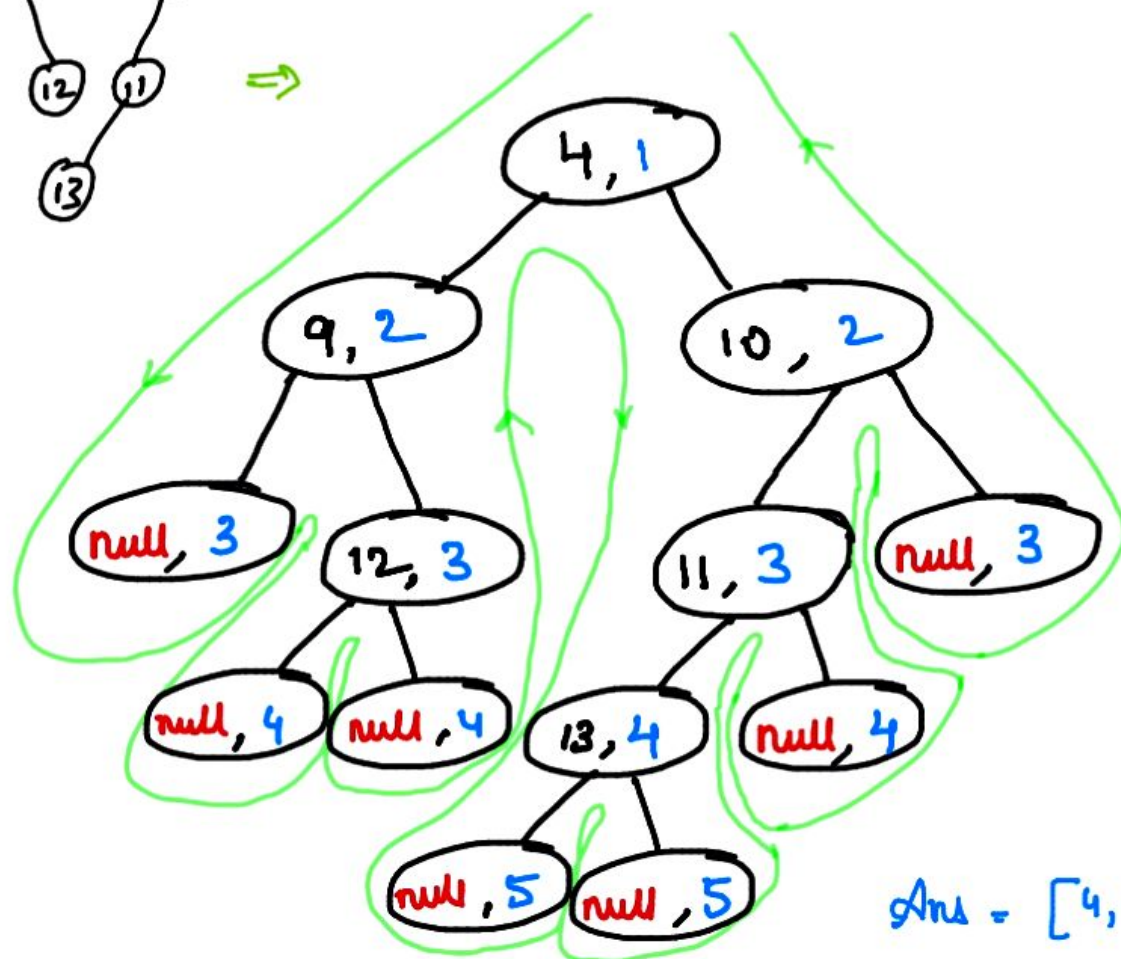
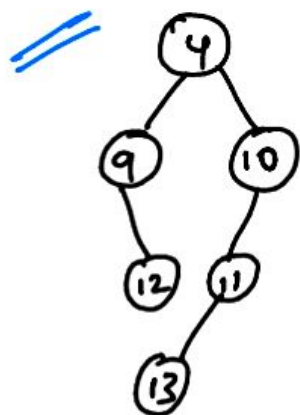
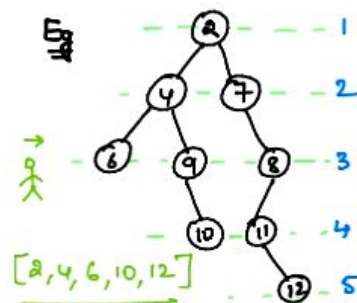
```
class Solution {
public:
    bool pathSumUtil(TreeNode* root, int currSum, int targetSum){
        if(root==NULL)
            return false;

        if(root->left==NULL && root->right==NULL){
            return (currSum+root->val)==targetSum;
        }

        return pathSumUtil(root->left, currSum+root->val, targetSum)
            || pathSumUtil(root->right, currSum+root->val, targetSum);
    }

    bool hasPathSum(TreeNode* root, int targetSum) {
        return pathSumUtil(root, 0, targetSum);
    }
};
```

# **D6** (15) Left view of a Binary Tree



Ans = [4, 9, 12, 13]

→ For every level traversed,  
check if it already exist in the set,

if already exist then continue,  
else add the root's value  
to array & into the set

Tc  $\rightarrow O(n)$

Sc  $\rightarrow O(n) + O(n) + O(h)$

↓  
result

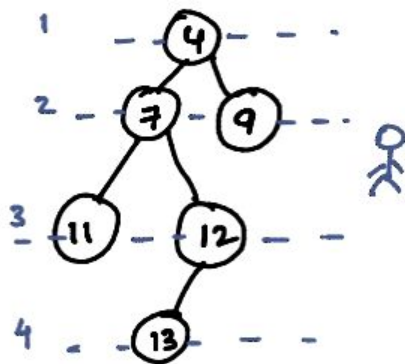
Code →

```
void viewGenerator(Node *root, vector<int> &res, set<int> &s, int currLevel){
    if(root==NULL) return;
    // if level is not reached, then add to result and the set
    if(s.find(currLevel)==s.end()){
        s.insert(currLevel);
        res.push_back(root->data);
    }
    // traverse the remaining branches
    viewGenerator(root->left, res, s, currLevel+1);
    viewGenerator(root->right, res, s, currLevel+1);
    return;
}

vector<int> leftView(Node *root)
{
    vector<int> res;
    set<int> s;
    viewGenerator(root, res, s, 0);
    return res;
}
```



## ⑩ Right view of Binary Tree →



Result = [4, 9, 12, 13]

→ The entire approach to solve the problem is same as the left view of binary tree. Even the time complexities.

→ only order of calling the branches change.

① right

② left

### code

```
class Solution {
public:
    void viewGenerator (TreeNode* root, vector<int> &res, set<int> &s, int currLevel){
        if(root==NULL) return;
        // if level is not reached, then add to result and the set
        if(s.find(currLevel)==s.end()){
            s.insert(currLevel);
            res.push_back(root->val);
        }
        // traverse the remaining branch
        viewGenerator(root->right, res, s, currLevel+1);
        viewGenerator(root->left, res, s, currLevel+1);
        return;
    }
    vector<int> rightSideView(TreeNode* root) {
        vector<int> res;
        set<int> s;
        viewGenerator(root, res, s, 0);
        return res;
    }
};
```

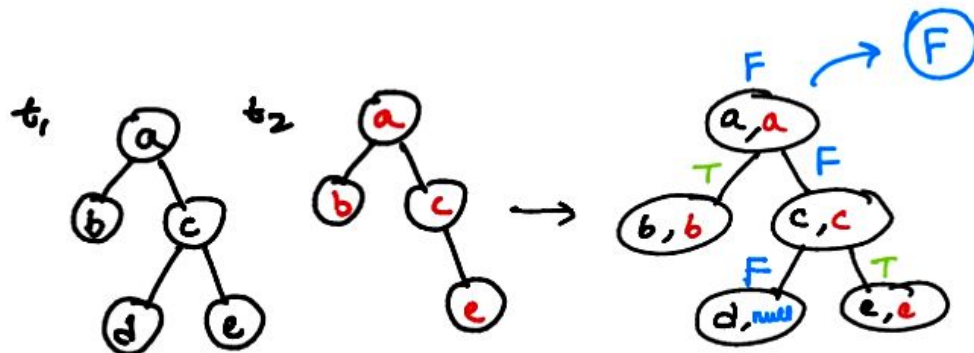
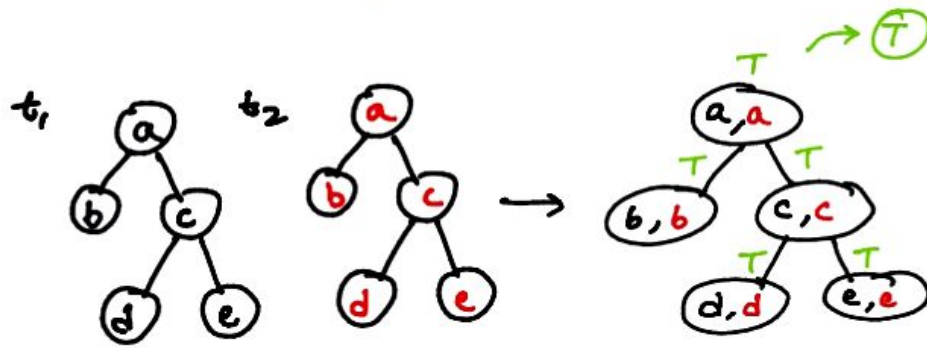
$T_c \rightarrow O(n)$

$S_c \rightarrow O(n) + O(n) + O(h)$

↓  
result



①7 Same tree → return true if both trees are same else false



$$T_c \rightarrow O(\min(m, n))$$

$$S_c \rightarrow O(1) + O(\min(h_1, h_2))$$

code →

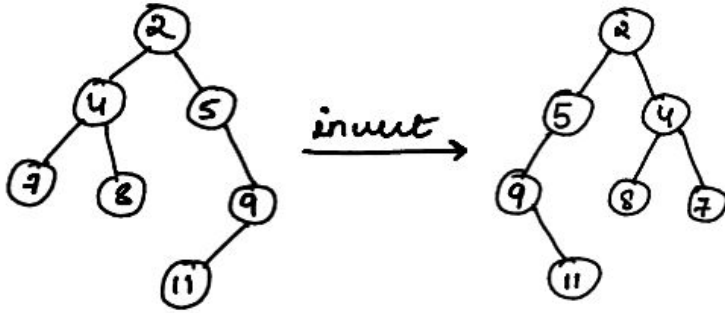
```
class Solution {
public:
    bool isSameTree(TreeNode* p, TreeNode* q) {
        if(p==NULL && q==NULL) return true;

        if(p==NULL || q==NULL || p->val != q->val) return false;

        return isSameTree(p->left, q->left) && isSameTree(p->right, q->right);
    }
};
```

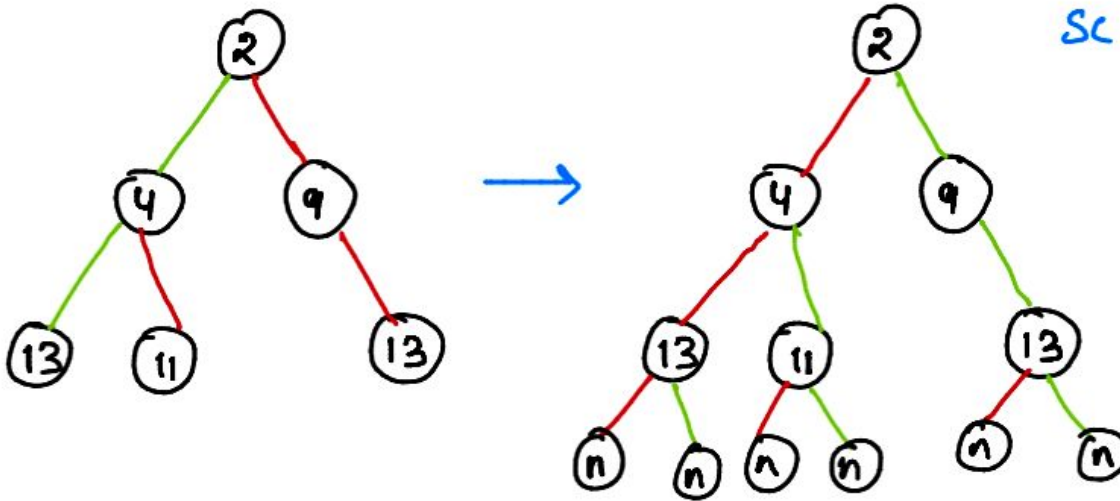
(18) Invert Binary Tree → given the root of BT, find its mirroring.

15



$T_c \rightarrow O(n)$

$S_c \rightarrow O(n) + O(h)$



Code →

```
class Solution {
public:
    TreeNode* invertTree(TreeNode* root) {
        if(root==NULL) return root;

        /* invert the left and right sub-trees and store
           them separately */
        TreeNode *leftSub = invertTree(root->right);
        TreeNode *rightSub = invertTree(root->left);

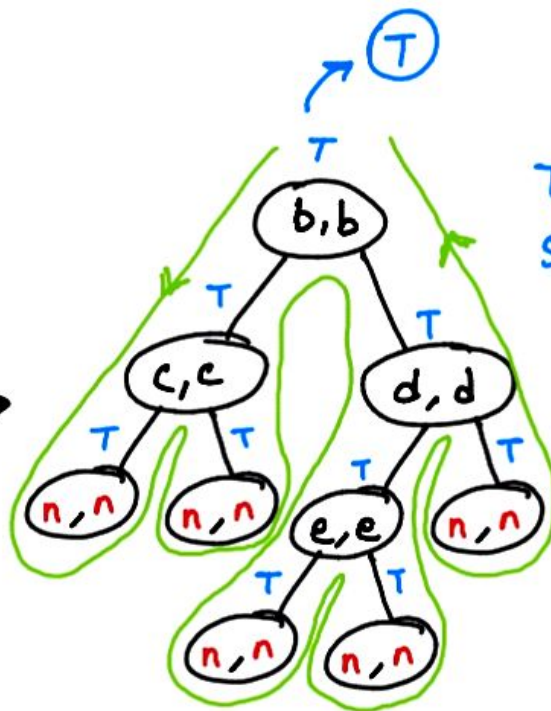
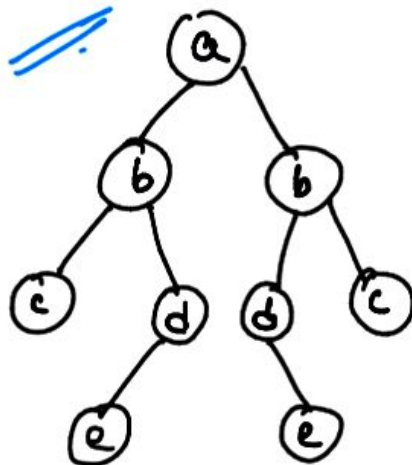
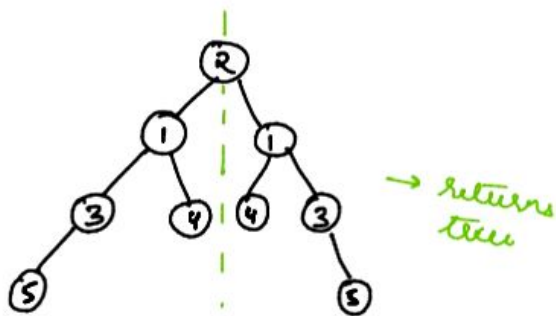
        // attach the branches to root
        root->left = leftSub;
        root->right = rightSub;

        return root;
    }
};
```

D7

# Q9 Symmetric Tree →

return true if left subtree is equal to right subtree, else return false



$T_c \rightarrow O(n)$   
 $S_c \rightarrow O(1) + O(h)$

code →

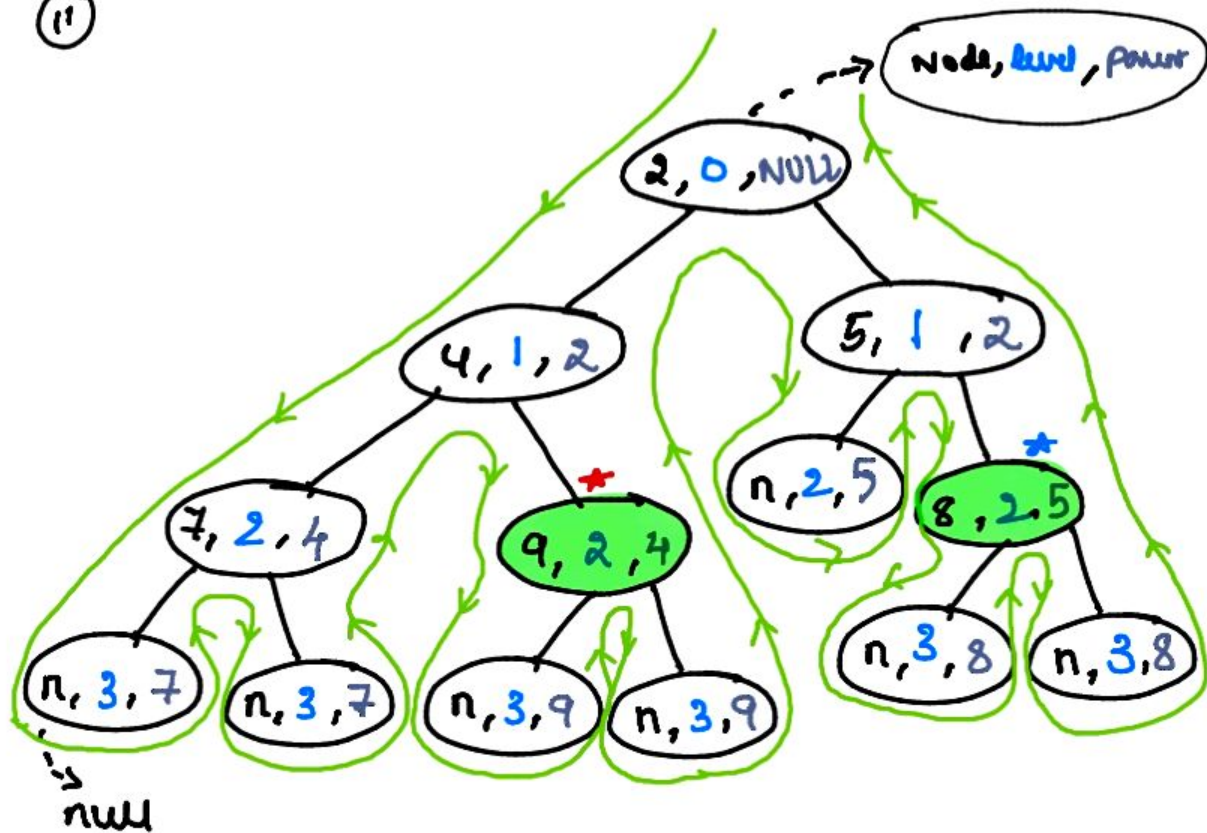
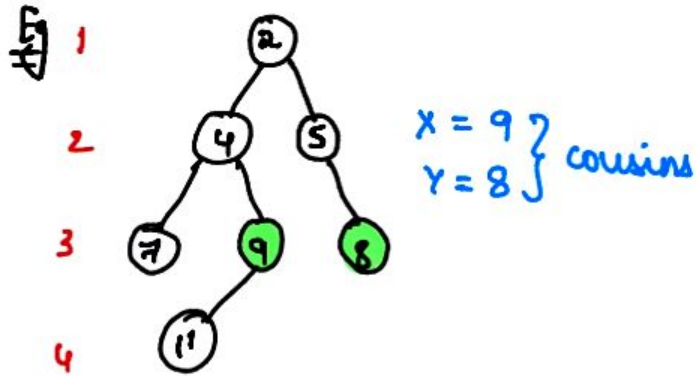
```
class Solution {
public:
    bool isMirror(TreeNode* l, TreeNode* r){
        if(l == NULL && r == NULL)
            return true;
        else if(l == NULL || r == NULL)
            return false;
        else if(l->val != r->val)
            return false;

        return isMirror(l->left, r->right) && isMirror(l->right, r->left);
    }
    bool isSymmetric(TreeNode* root) {
        if(root == NULL) return true;
        return isMirror(root->left, root->right);
    }
};
```



20) Cousins of a Binary Tree → given two nodes, find if they are cousins of each other.

same level but diff parents.



\* at this step as value = 9 is

x is found store its parent & level in separate variables

\* later compare its value with other occurrence in Y such that

- 1)  $x.parent \neq y.parent$
- 2)  $x.level == y.level$

$TC \rightarrow O(n)$

$SC \rightarrow O(1)$

Recursive  
Stack  $\rightarrow O(n)$



## Code

```
class Solution {
public:
    void findNodes(TreeNode* root, int x, int y, int level[2], int parents[2], int currlevel, TreeNode* currparent)
    {
        if(root==NULL) return;
        if(root->val == x)
        {
            level[0]=currlevel;
            parents[0]=currparent->val;
        }
        if(root->val == y)
        {
            level[1]=currlevel;
            parents[1]=currparent->val;
        }
        findNodes(root->left, x, y, level, parents, currlevel+1, root);
        findNodes(root->right, x, y, level, parents, currlevel+1, root);
    }
    bool isCousins(TreeNode* root, int x, int y) {
        int level [2] = {-1,-1};
        int parents[2] = {-1,-1};
        findNodes(root, x, y, level, parents, 0, new TreeNode(-1));
        if(level[0]==level[1] && parents[0]!=parents[1])
            return true;
        return false;
    }
};
```

# Trees - Part 2

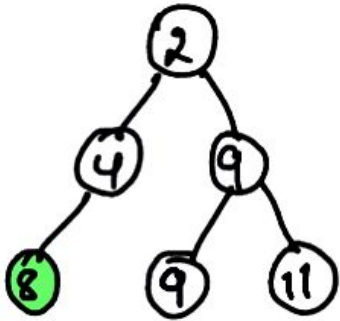
- Karun Karthik

## Contents

21. Print all nodes that do not have any siblings
22. All nodes at distance K in a Binary Tree
23. Lowest Common Ancestor
24. Level order traversal in Binary Tree
25. Level order traversal in N-ary Tree
26. Top view of Binary Tree
27. Bottom view of Binary Tree
28. Introduction to Binary Search Tree & Search in a BST
29. Insert into a BST
30. Range Sum of BST
31. Increasing order search tree
32. Two Sum IV
33. Delete Node in a BST
34. Inorder successor in BST
35. Validate BST
36. Lowest Common Ancestor of BST
37. Convert Sorted Array to BST
38. Construct BT from Preorder and Inorder traversal
39. Construct BT from Inorder and Postorder traversal
40. Construct BST from Preorder traversal

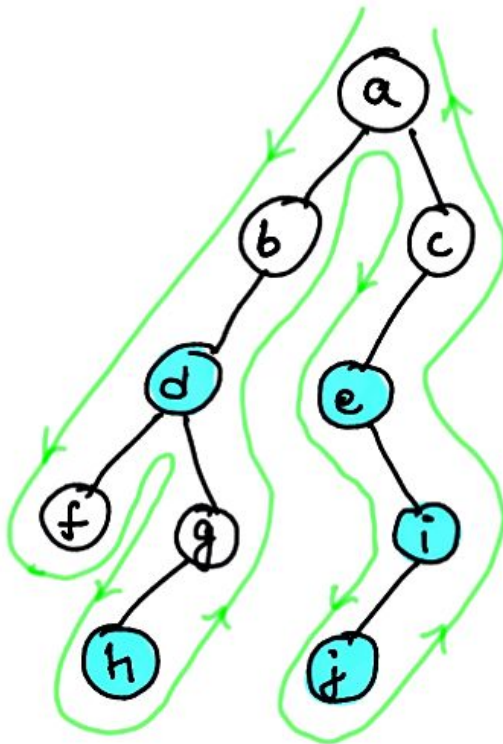
D7 (21) Print all nodes that don't have any siblings

LSA



Result ↗

Sibling  $\rightarrow$  same level, same parent



$T_c \rightarrow O(n)$

$S_c \rightarrow O(n)$

At every node, check if

both branches exist  $\rightarrow$  then call both of them recursively

only left branch exist  $\rightarrow$  then call left branch recursively

only right branch exist  $\rightarrow$  then call right branch recursively

Code →

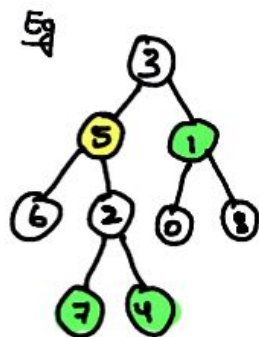
```
1 void findNode(Node* root, vector<int>&res){
2
3     if(root==NULL) return;
4     if(root->left == NULL && root->right==NULL) return;
5
6     // both branches present then call recursively
7     if(root->left!=NULL && root->right!=NULL)
8     {
9         findNode(root->left, res);
10        findNode(root->right, res);
11    }
12    else if(root->left!=NULL) // right branch absent
13    {
14        res.push_back(root->left->data);
15        findNode(root->left, res);
16
17    } else if(root->right!=NULL) // left branch absent
18    {
19        res.push_back(root->right->data);
20        findNode(root->right, res);
21    }
22    return;
23 }
24
25 vector<int> noSibling(Node* node)
26 {
27     vector<int> res;
28     findNode(node, res);
29     if(res.size()==0) res.push_back(-1);
30     sort(res.begin(), res.end());
31     return res;
32 }
33
```



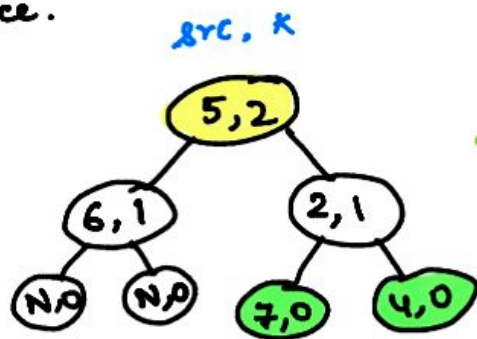
# D8 (22) All nodes distance K in Binary Tree

Given a source node, find all the nodes that are at a distance of K units.

① consider nodes in downward direction of source.



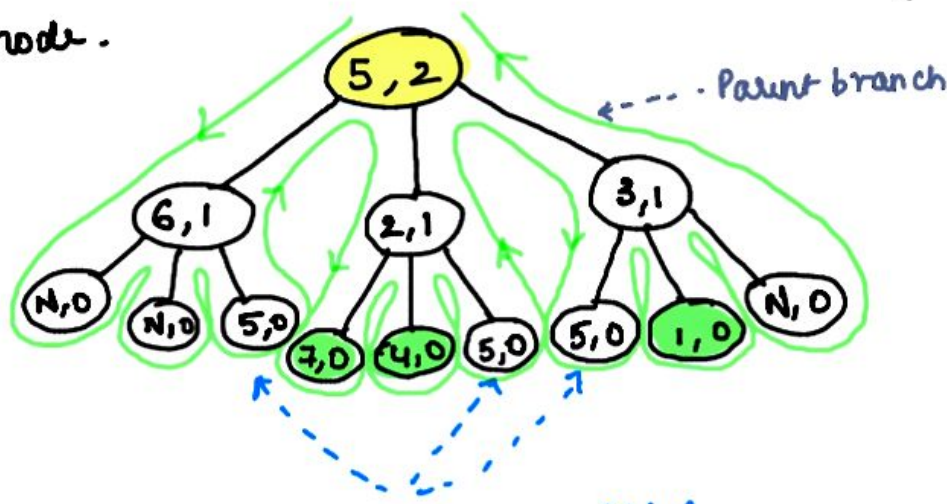
Source = 5  
K = 2



• Store in result if  $K=0$   
Return if  $K < 0$

② To solve for the upward direction we can use hashing to store the parent node.

Node	Parent
3	NULL
5	3
1	3
6	5
2	5
0	1
8	1
7	2
4	2



\* If  $K=0$   
& the node value is valid (not null) then add to result array

already visited

↳ to handle such case we use a set.

populating traversal

$$Tc \rightarrow O(n) + O(n)$$

$$Sc \rightarrow O(n) + O(n) + O(1)$$

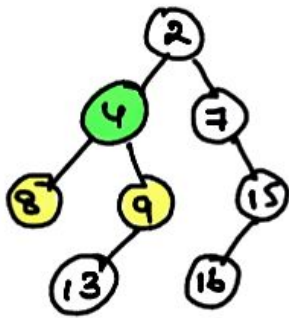
result

Code.

```
1  class Solution {
2  public:
3      // to create hashtable
4      void populateMap(TreeNode* currnode, TreeNode* currparent,
5                          unordered_map<TreeNode*,TreeNode*>&parentmap){
6          if(currnode == NULL) return;
7          parentmap[currnode] = currparent;
8          populateMap(currnode->left,currnode,parentmap);
9          populateMap(currnode->right,currnode,parentmap);
10         return;
11     }
12
13     // finding all the nodes at distance K
14     void printKdistance(TreeNode* currnode, int k, set<TreeNode*>&s,
15                         unordered_map<TreeNode*,TreeNode*>&parentmap, vector<int>&ans)
16     {
17         if(currnode == NULL || s.find(currnode)!=s.end() || k<0)
18             return;
19
20         s.insert(currnode);
21
22         if(k==0)
23         {
24             ans.push_back(currnode->val);
25             return;
26         }
27
28         printKdistance(currnode->left,k-1,s,parentmap,ans); // call left child
29         printKdistance(currnode->right,k-1,s,parentmap,ans); // call right child
30         printKdistance(parentmap[currnode],k-1,s,parentmap,ans); // call the parent
31         return;
32     }
33
34     vector<int> distanceK(TreeNode* root, TreeNode* target, int k) {
35         vector<int>ans;
36         set<TreeNode*>s;
37         unordered_map<TreeNode*,TreeNode*>parentmap;
38         populateMap(root,NULL,parentmap);
39         printKdistance(target,k,s,parentmap,ans);
40         return ans;
41     }
42 };
```

## (23) Lowest Common Ancestor

Eg



node to root paths  
↓  
 $n_1 = 8, n_2 = 9$  then  $\bar{n}_1 = [8, 4, 2]$   
 $\bar{n}_2 = [9, 4, 2]$  }  $\rightarrow 4$

$n_1 = 9, n_2 = 13$  then  $\bar{n}_1 = [9, 4, 2]$   
 $\bar{n}_2 = [13, 9, 4, 2]$  }  $\rightarrow 9$

→ for every node, check if it matches  $n_1$  or  $n_2$ .  
if found return node  
else call recursively in both branches.  
if both return non-null value  $\Rightarrow$  root is LCA  
else return the branch value that is non-null.

### code

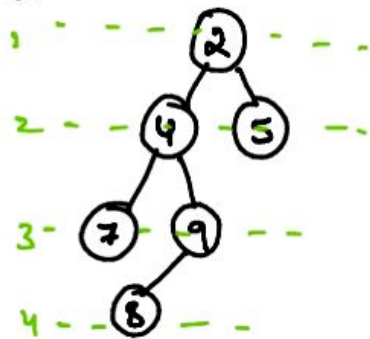
```
class Solution {
public:
    TreeNode* lowestCommonAncestor(TreeNode* root, TreeNode* p, TreeNode* q) {
        if(root==NULL) return NULL;
        if(root->val == p->val || root->val == q->val) return root;
        TreeNode* leftSubTree = lowestCommonAncestor(root->left, p, q);
        TreeNode* rightSubTree = lowestCommonAncestor(root->right, p, q);
        if(leftSubTree!=NULL && rightSubTree!=NULL) return root;
        if(leftSubTree!=NULL) return leftSubTree;
        if(rightSubTree!=NULL) return rightSubTree;
        return NULL;
    }
};
```



# D9 (24) Level order traversal Binary Tree

Given root node, find level order traversal.

Eg.



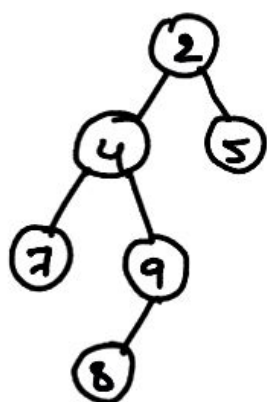
$\Rightarrow [ [2], [4, 5], [7, 9], [8] ]$

Tc  $\rightarrow O(n)$

Sc  $\rightarrow O(n)$

$\rightarrow$  To find level order traversal use queue. FIFO data structure  
 Inserting  $\rightarrow$  Rear  
 Removing  $\rightarrow$  Front

$\rightarrow$  Before removing from queue, add the children to the queue (BFS)



Queue

2

4, 5

5, 7, 9

7, 9

9

8

empty  $\rightarrow$

Result

[ ]

[ 2 ]

[ 2, 4 ]

[ 2, 4, 5 ]

[ 2, 4, 5, 7 ]

[ 2, 4, 5, 7, 9 ]

Answer  $\rightarrow [ 2, 4, 5, 7, 9, 8 ]$

① For every node, enqueue.

② while dequeue, enqueue its branches.



Code →

```
1  class Solution {
2  public:
3      vector<vector<int>> levelOrder(TreeNode* root) {
4          vector<vector<int>> res;
5          queue<TreeNode*> q;
6
7          if(root==NULL) return res;
8          q.push(root);
9
10         while(!q.empty()){
11
12             int currsz = q.size();
13             vector<int> currLevel;
14
15             while(currsz>0)
16             {
17                 TreeNode* currnode = q.front();
18                 q.pop();
19                 currLevel.push_back(currnode->val);
20                 currsz--;
21
22                 if(currnode->left!=NULL)
23                     q.push(currnode->left);
24
25                 if(currnode->right!=NULL)
26                     q.push(currnode->right);
27             }
28             res.push_back(currLevel);
29         }
30         return res;
31     }
32 };
```

## 25) Level order traversal N-ary Tree

→ Everything is same as previous problem, intuition & complexity

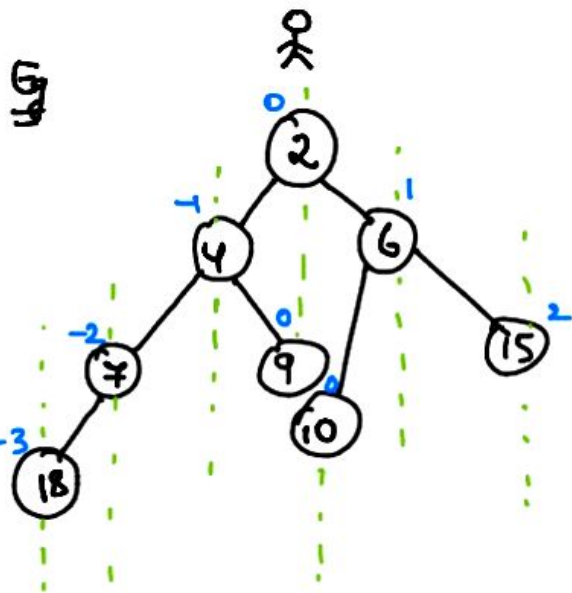
$T_c \rightarrow O(n)$

$S_c \rightarrow O(n)$

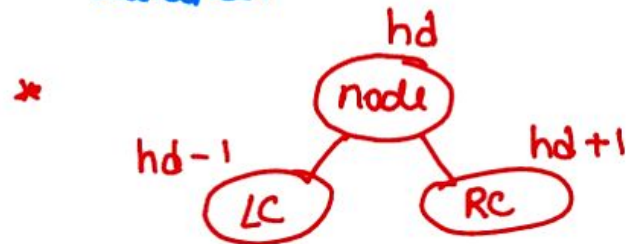
code →

```
1  class Solution {
2  public:
3      vector<vector<int>> levelOrder(Node* root) {
4          vector<vector<int>> res;
5          queue<Node*> q;
6
7          if(root == NULL) return res;
8          q.push(root);
9
10         while(!q.empty())
11         {
12             int currsz = q.size();
13             vector<int> currLevel;
14             while(currsz > 0)
15             {
16                 Node* currnode = q.front();
17                 q.pop();
18                 currLevel.push_back(currnode->val);
19                 currsz--;
20
21                 // enqueue all the children
22                 for(auto child: currnode->children)
23                     q.push(child);
24             }
25             res.push_back(currLevel);
26         }
27         return res;
28     }
29 };
```

## (26) Top View of Binary Tree



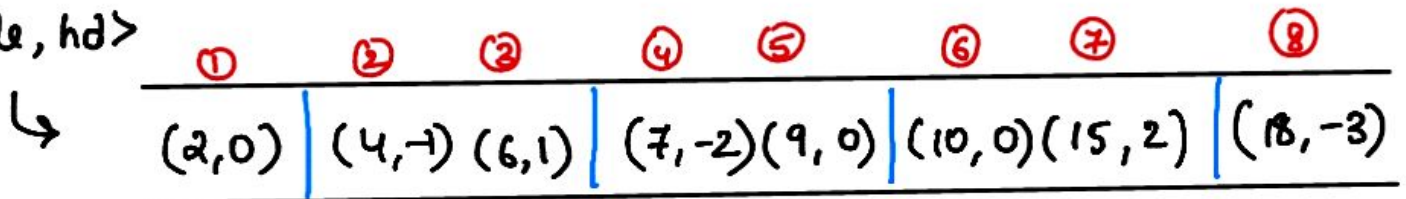
\* For top view or bottom view we use concept of horizontal distance.



\*  $hd \text{ of root} = 0$

\* make a pair with node & its  $hd$ .  
& perform bfs.

$\langle \text{node}, hd \rangle$



use a hashmap to store result.

HD	NODE
0	2
-1	4
1	6
-2	7
2	15
-3	18

- ① As  $hd = 0$  is not present in map add 2 to map.
- ② As  $hd = -1$  is not present in map add 4 to map.
- ③ As  $hd = 1$  is not present in map add 6 to map.
- ④ As  $hd = -2$  is not present in map add 7 to map.
- ⑤  $hd = 0$  is already present.
- ⑥  $hd = 0$  is already present.
- ⑦ As  $hd = 2$  is not present in map add 15 to map.
- ⑧ As  $hd = -3$  is not present in map add 18 to map.

→ convert into array & return as result.

## code

```
1  class Solution
2  {
3      public:
4      vector<int> topView(Node *root)
5      {
6          vector<int> res;
7          if(root==NULL) return res;
8
9          map<int,int> mp;
10         queue<pair<Node*,int>> q;
11
12         q.push({root,0});
13
14         while(!q.empty()){
15
16             auto it = q.front();
17             q.pop();
18
19             Node* node = it.first;
20             int hd = it.second;
21
22             if(mp.find(hd) == mp.end())
23                 mp[hd] = node->data;
24
25             if(node->left!=NULL)
26                 q.push({node->left,hd-1});
27
28             if(node->right!=NULL)
29                 q.push({node->right,hd+1});
30         }
31
32         // store in vector or array
33         for(auto it:mp)
34             res.push_back(it.second);
35
36         return res;
37     }
38 };
39
40
```

$\log n \rightarrow \text{map}$

$T_c \rightarrow O(n \log n)$

$S_c \rightarrow O(n)$



## (27) Bottom View of Binary Tree

→ Similar to top view, but replace entries in hashmap so you'll get last possible element with particular hd.

Code →

```
1  class Solution {
2      public:
3          vector<int> bottomView(Node *root) {
4              vector<int> res;
5              if(root==NULL) return res;
6
7              map<int, int> mp;
8              queue<pair<Node*, int>>q;
9
10             q.push({root, 0});
11             while(!q.empty()){
12                 auto it = q.front();
13                 q.pop();
14
15                 Node* node = it.first;
16                 int hd = it.second;
17
18                 mp[hd] = node->data;
19
20                 if(node->left!=NULL)
21                     q.push({node->left, hd-1});
22
23                 if(node->right!=NULL)
24                     q.push({node->right, hd+1});
25             }
26
27             for(auto it:mp)
28                 res.push_back(it.second);
29
30             return res;
31         }
32     };
```

D10

## Binary Search Tree

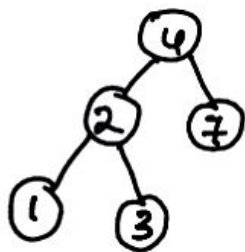
- every node is  $>$  than previous node &  $<$  than next node.
- If duplicates, then it'll be mentioned that it'll be included in LC or RC

$$(1) \quad LC < node < RC$$

$$(2) \quad LC \leq node < RC$$

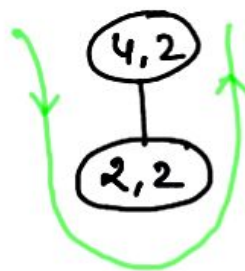
$$(3) \quad LC < node \leq RC$$

### (28) Search in a BST



val = 2.

⇒ return the subtree with given value.



• as  $2 < 4$ , search in LST.

• as  $2 == 2$  return node.

TC  $\rightarrow O(\log_2 n)$  ,  $O(n)$   
                     avg                      worst

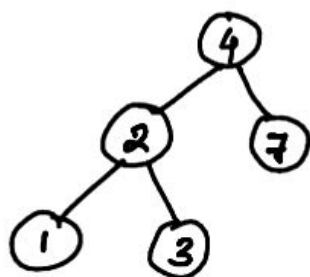
SC  $\rightarrow O(n)$

### code

```

class Solution {
public:
    TreeNode* searchBST(TreeNode* root, int val) {
        if(root == NULL) return NULL;
        if(root->val == val) return root;
        if(root->val < val) return searchBST(root->right, val);
        return searchBST(root->left, val);
    }
};
  
```

## 29) Insert into BST



val = 5.

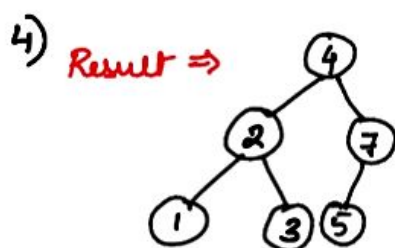
TC  $\rightarrow O(\log_2 n)$  ,  $O(n)$   
            avg              worst

SC  $\rightarrow O(1)$

Code  $\rightarrow$

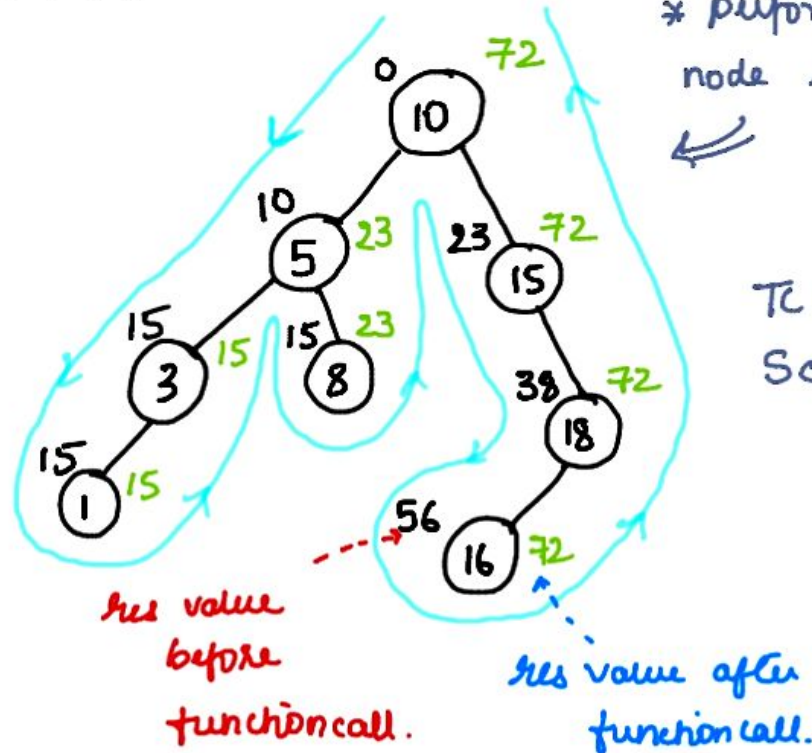
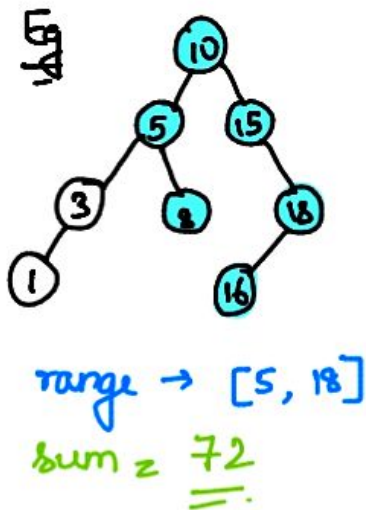
```
class Solution {
public:
    TreeNode* insertIntoBST(TreeNode* node, int val) {
        if(node==NULL){
            return new TreeNode(val);
        }
        if (val < node->val) {
            node->left = insertIntoBST(node->left, val);
        }
        else {
            node->right = insertIntoBST(node->right, val);
        }
        return node;
    }
};
```

- 1) <sup>root</sup> 4 5 As  $5 > 4$ , go to RST
- 2) 4 5 7 As  $5 < 7$ , go to LST
- 3) • As LST of 7 is null, create node with value = 5. 5  
• Link 5 as LST of 7.



### 30) Range sum of BST

Given a root node & interval  $[x, y]$ , find sum of all nodes that lies in  $[x, y]$ .



TC  $\rightarrow O(n)$   
SC  $\rightarrow O(n)$

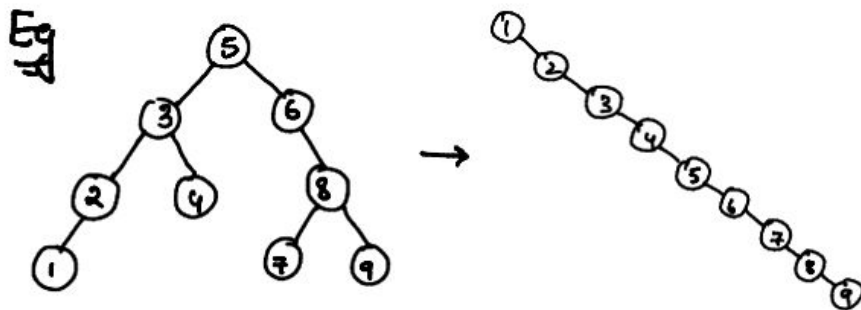
Code  $\rightarrow$

```
1 class Solution {
2 public:
3     void sumUtil(TreeNode* root, int low, int high, int &res){
4         if(root==NULL) return;
5         if(root->val <= high && root->val >= low){
6             res += root->val;
7         }
8         sumUtil(root->left, low, high, res);
9         sumUtil(root->right, low, high, res);
10    }
11
12    int rangeSumBST(TreeNode* root, int low, int high) {
13        int res = 0;
14        sumUtil(root, low, high, res);
15        return res;
16    }
17 };
```



## ③1 Increasing order search tree

Given a BST, create an increasing order search tree.



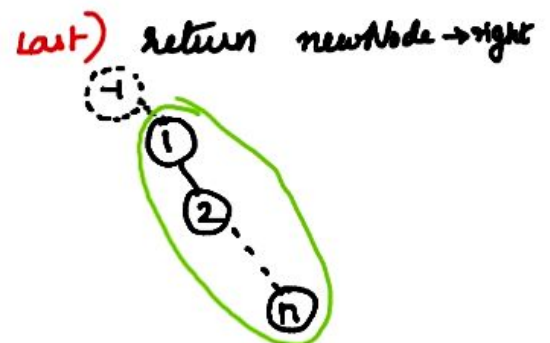
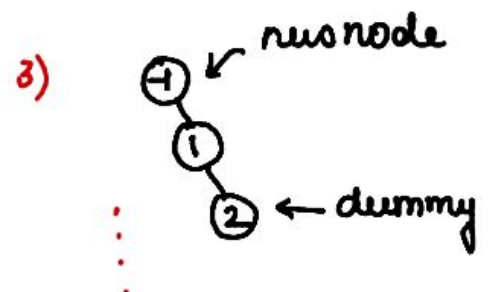
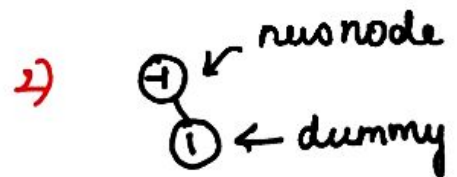
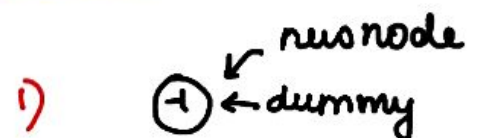
① Perform inorder traversal.

② create a skewed tree using elements in inorder traversal.

## Code

```
1 class Solution {
2 public:
3     void inorder(TreeNode* root, vector<int> &res){
4         if(root==NULL) return;
5         inorder(root->left, res);
6         res.push_back(root->val);
7         inorder(root->right, res);
8     }
9     TreeNode* increasingBST(TreeNode* root) {
10         vector<int> res;
11         inorder(root, res);
12
13         // create right skewed tree
14         TreeNode* dummy = new TreeNode(-1);
15         TreeNode* newNode = dummy;
16         for(auto it: res){
17             dummy->right = new TreeNode(it);
18             dummy = dummy->right;
19         }
20         return newNode->right;
21     }
22 };
```

## Lines 16-20



32

↳ return true if sum of any 2 values == k


$$k = 9$$
 $\Rightarrow$ 

as  $v[f] + v[r] = k$ , return true, else  $f++$  or  $r--$   
as per sum &  $k$ .

code →

```
class Solution {
public:
    void inorder(TreeNode* root, vector<int> &res){
        if(root==NULL) return;
        inorder(root->left, res);
        res.push_back(root->val);
        inorder(root->right, res);
    }
    bool findTarget(TreeNode* root, int k) {
        vector<int> res;
        inorder(root, res);
        int front = 0;
        int rear = res.size()-1;
        while(front<rear){
            if(res[front]+res[rear]==k) return true;
            if(res[front]+res[rear]>k) rear--;
            else front++;
        }
        return false;
    }
};
```

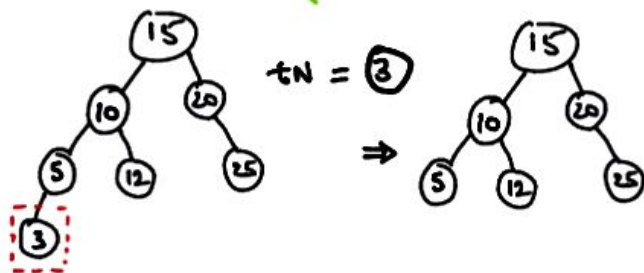
$$T_c \rightarrow O(n) + O(n)$$
$$S_c \rightarrow O(n)$$

# D11 (33) Delete Node in BST

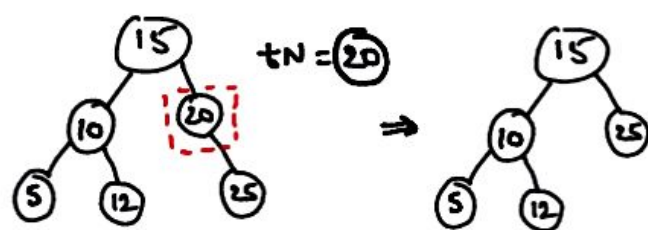
Given root of BST & a target node, delete the target node & return the tree.

Cases →

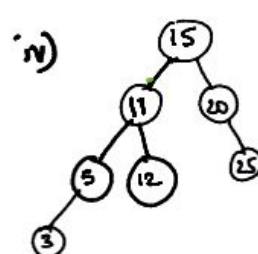
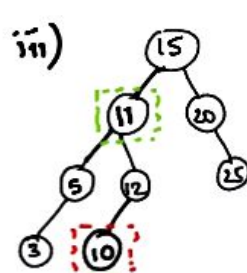
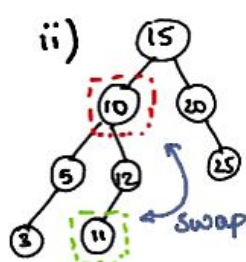
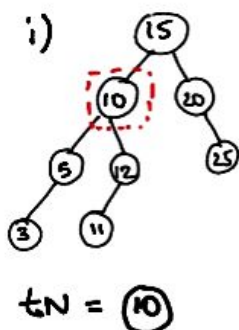
- ① If target node is leaf →  
then simply delete it



- ② If target node has 1 child →  
then remove node & return the subtree



- ③ If target node has 2 children →  
then go to right child's left subtree & swap its value with target node & then delete it.



```
1  class Solution {
2  public:
3      TreeNode* findleftmostNode(TreeNode* root){
4          while(root->left!=NULL)
5              root = root->left;
6          return root;
7      }
8
9      TreeNode* deleteNode(TreeNode* root, int key) {
10
11          if(root==NULL) return NULL;
12
13          if(root->val > key)
14              root->left = deleteNode(root->left, key);
15
16          else if(root->val < key)
17              root->right = deleteNode(root->right, key);
18
19          else { // root->val == key
20              if(root->left == NULL && root->right == NULL){
21                  root = NULL;
22                  return root;
23              }
24              if(root->left != NULL && root->right == NULL){
25                  root = root->left;
26                  return root;
27              }
28              if(root->right != NULL && root->left == NULL){
29                  root = root->right;
30                  return root;
31              }
32
33              // finding left most node in right subtree
34              TreeNode* temp = findleftmostNode(root->right);
35
36              //swapping root's value with left most node's val
37              int tempVal = root->val;
38              root->val = temp->val;
39              temp->val = tempVal;
40
41              // performing delete in right subtree
42              root->right = deleteNode(root->right, key);
43              return root;
44          }
45          return root;
46      }
47  };
```

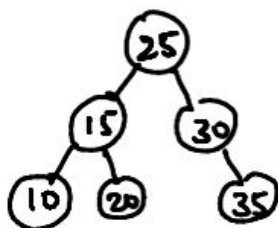


### (34) Inorder successor of BST

Given root, find inorder successor of given node

↳ the element just after the node in inorder traversal.

eg



n = 15    o/p → 20

n = 35    o/p → null.

Code →

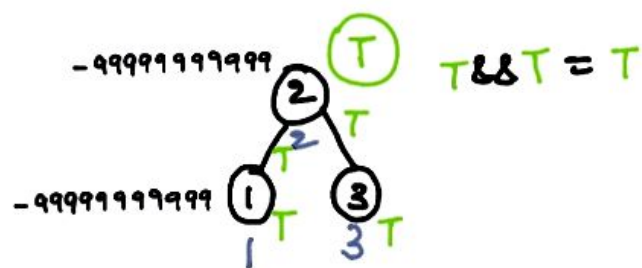
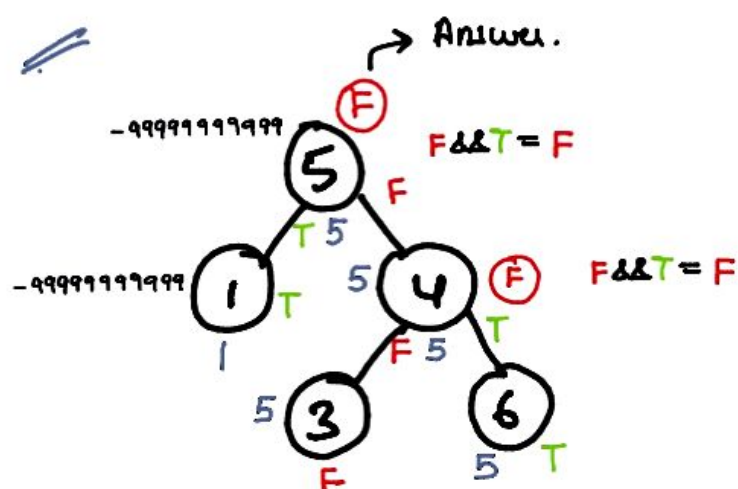
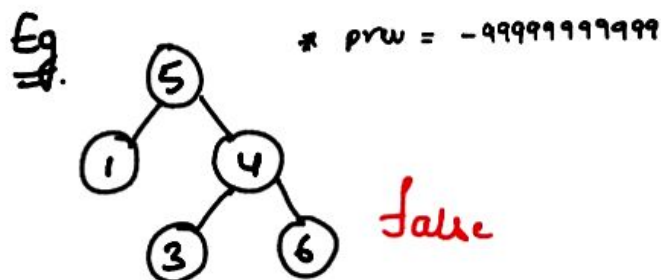
```
class Solution{
public:

    void inorder(Node *root, vector<Node*> &res){
        if(root==NULL) return;
        inorder(root->left, res);
        res.push_back(root);
        inorder(root->right, res);
    }

    Node * inOrderSuccessor(Node *root, Node *x)
    {
        vector<Node*> res;
        inorder(root, res);
        for(int i=0; i<res.size(); i++){
            if(res[i]==x && i<res.size()-1){
                return res[i+1];
            }
        }
        return NULL;
    }
};
```

D12 (35) Validate BST given a root node, return true if it is valid BST

\* Every value should be less than previous one in Inorder traversal



- Return True on NULL nodes
- Check for left subtree
- previous value gets updated before checking Right subtree & after checking left subtree
- if currVal <= previous then return false
- return true if both LST & RST are BST

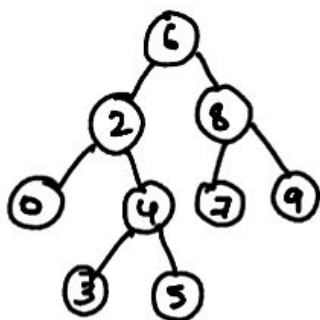
code

```
class Solution {
public:
    bool isBST(TreeNode* root, long int &prev){
        if(root==NULL) return true;
        bool isLeftBalanced = isBST(root->left, prev);
        if(root->val <= prev) return false;
        prev = root->val;
        bool isRightBalanced = isBST(root->right, prev);
        return isLeftBalanced && isRightBalanced;
    }

    bool isValidBST(TreeNode* root) {
        long int prev = -9999999999;
        return isBST(root, prev);
    }
};
```

### 36) LCA of BST →

Eg.



$p=2, q=8$

if  $currNode > \text{both } p \text{ \& } q$   
then LCA lies in LST

if  $currNode < \text{both } p \text{ \& } q$   
then LCA lies in RST

in every other case the  $currNode$  is  
LCA as  $p \text{ \& } q$  will be on

	worst	avg
Tc →	$O(n)$	$O(\log n)$
Sc →	$O(n)$	

code

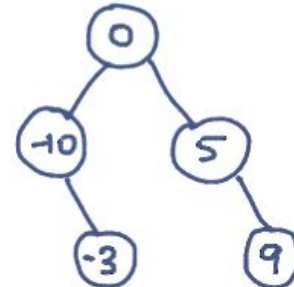
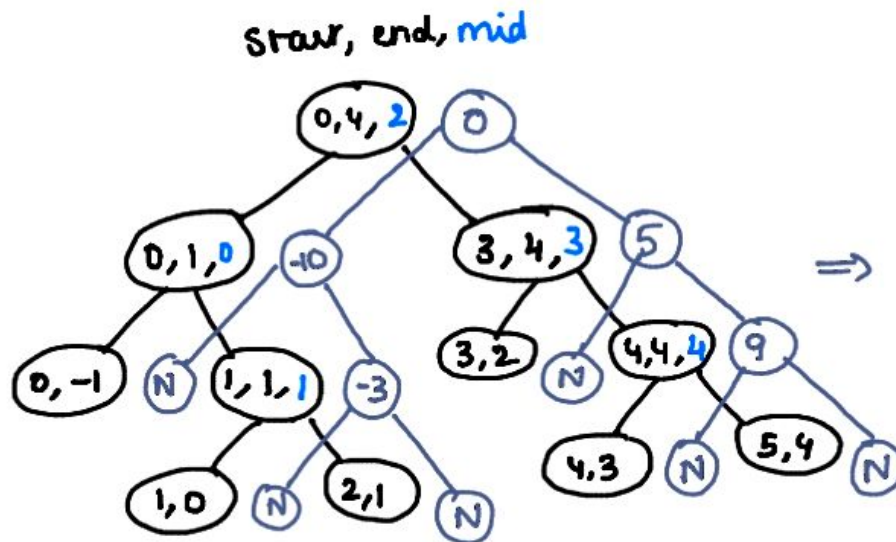
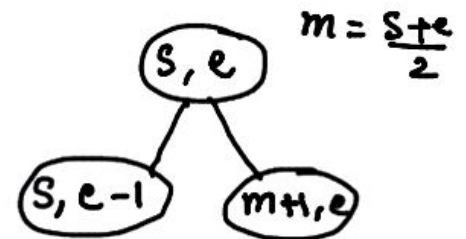
```
class Solution {
public:
    TreeNode* lowestCommonAncestor(TreeNode* root, TreeNode* p, TreeNode* q) {
        if(root==NULL) return NULL;

        if(root->val < p->val && root->val < q->val){
            return lowestCommonAncestor(root->right, p, q);
        }
        else if(root->val > p->val && root->val > q->val){
            return lowestCommonAncestor(root->left, p, q);
        }
        else {
            return root;
        }
    }
};
```

### 37) Convert Sorted array to BST

Given sorted array, create a BST

Eg  $[-10, -3, 0, 5, 9]$   
0 1 2 3 4



code →

```
class Solution {
public:
    TreeNode* createBST(vector<int>&nums, int start, int end){
        if(start > end) return NULL;

        int mid = (start + end)/2;
        TreeNode* root = new TreeNode(nums[mid]);

        root->left = createBST(nums, start, mid-1);
        root->right = createBST(nums, mid+1, end);
        return root;
    }

    TreeNode* sortedArrayToBST(vector<int>& nums) {
        return createBST(nums, 0, nums.size()-1);
    }
};
```



# DI3 (38) Construct Binary Tree from Pre & Inorder traversal

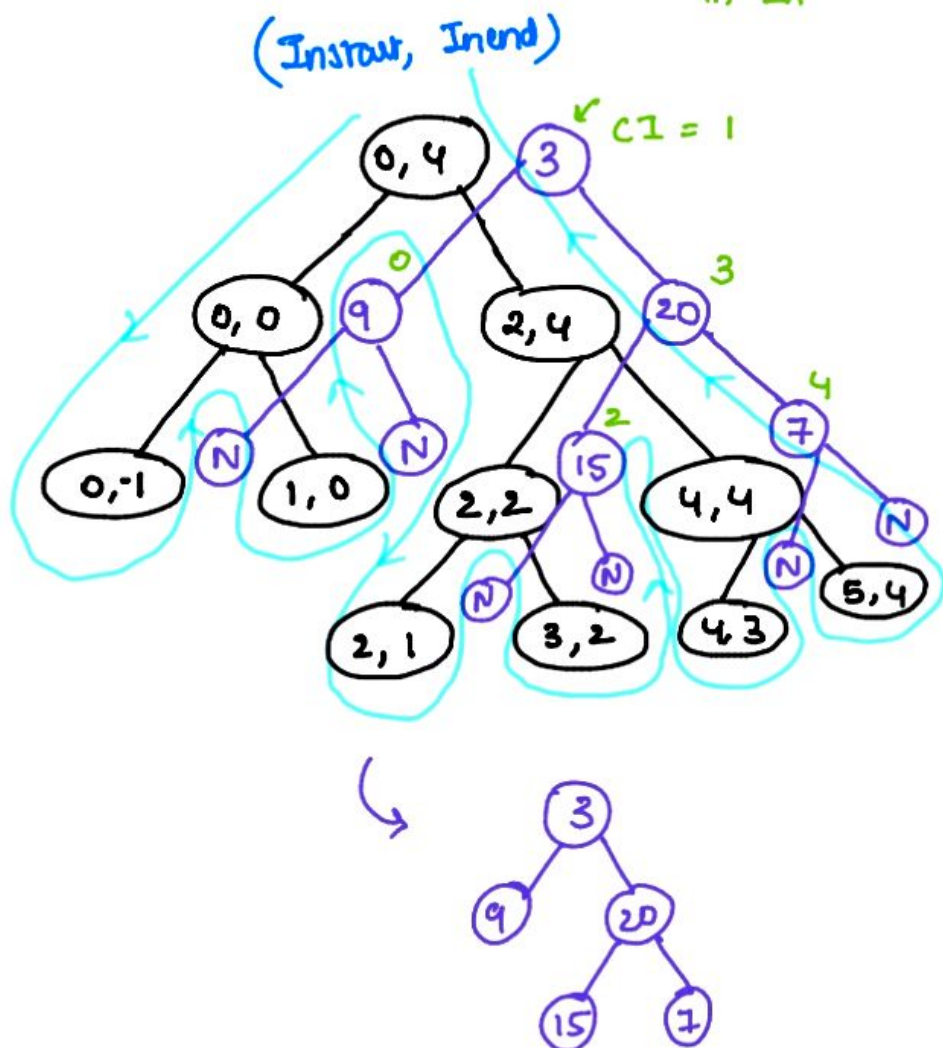
Ex  
 $pre = [3, 9, 20, 15, 7]$   
 $in = [9, 3, 15, 20, 7]$

\* for every node in Pre, the corresponding LST & RST are in In

ie  $3 \rightarrow [9, 3, 15, 20, 7]$   
 LST CI RST  
 0 1 2 3 4  
 9 3 15 20 7

✓  
 $LST = (instart, CI-1)$   
 $RST = (CI+1, inend)$

CI = index of pre[0]  
 in In



$TC \rightarrow O(n^2)$

$SC \rightarrow O(n)$

$pre = [3, 9, 20, 15, 7]$   
 $in = [9, 3, 15, 20, 7]$

- for preorder index = 0, inorder boundary = [0, 4]
- find root value in Inorder array & its index is CurrIndex
- if  $instart > CI-1$  or  $CI+1 < inend$  return NULL

To reduce TC  
 we can use  
 hashtable to find  
 indexing

$TC \rightarrow O(n)$

$SC \rightarrow O(n) + O(n)$

code →

```
1  class Solution {
2  public:
3      TreeNode* constructTree(vector<int>& preorder, unordered_map<int, int> &mp,
4      int start, int end, int &preIdx ){
5
6          if(start>end)    return NULL;
7          TreeNode* root = new TreeNode(preorder[preIdx]);
8
9          // find currIndex as per inorder array
10         int currIdx = mp[preorder[preIdx]];
11         // increment preIdx to find next root
12         preIdx++;
13
14         // recursively call LST & RST
15         root->left = constructTree(preorder, mp, start, currIdx-1, preIdx);
16         root->right = constructTree(preorder, mp, currIdx+1, end, preIdx);
17         return root;
18     }
19
20     unordered_map<int,int> populate(vector<int>&inorder){
21         unordered_map<int,int> mp;
22         for(int i=0; i<inorder.size(); i++){
23             mp[inorder[i]] = i;
24         }
25         return mp;
26     }
27
28     TreeNode* buildTree(vector<int>& preorder, vector<int>& inorder) {
29         unordered_map<int,int> mp = populate(inorder);
30         int preIdx = 0;
31         return constructTree(preorder, mp, 0, inorder.size()-1, preIdx);
32     }
33 };
34
```

### ③⑨ Construct Binary Tree from In & Postorder Traversal

Intuition is same as previous program, only changes are

- traverse from last element in postorder array
- process RST & then go for LST

Code →

```
1  class Solution {
2  public:
3
4      TreeNode* constructTree(vector<int>& postorder, unordered_map<int, int> &mp,
5                              int start, int end, int &postIdx){
6
7          if(start>end)    return NULL;
8          TreeNode* root = new TreeNode(postorder[postIdx]);
9
10         // find currIndex as per inorder array
11         int currIdx = mp[postorder[postIdx]];
12         postIdx--;
13
14         // recursively call RST & LST
15         root->right = constructTree(postorder, mp, currIdx+1, end, postIdx);
16         root->left = constructTree(postorder, mp, start, currIdx-1, postIdx);
17         return root;
18     }
19
20     unordered_map<int,int> populate(vector<int>&inorder){
21         unordered_map<int,int> mp;
22         for(int i=0; i<inorder.size(); i++){
23             mp[inorder[i]] = i;
24         }
25         return mp;
26     }
27
28     TreeNode* buildTree(vector<int>& inorder, vector<int>& postorder) {
29         unordered_map<int,int> mp = populate(inorder);
30         int postIdx = postorder.size()-1;
31         return constructTree(postorder, mp, 0, inorder.size()-1, postIdx);
32     }
33 };
```



# (40) Construct BST from Preorder traversal

[8, 5, 1, 7, 10, 12]

$T_c \rightarrow O(n \log n)$  (due to sorting)

Approach 1  $\rightarrow$  Sort given Preorder to get Inorder, now similar to problem (28).

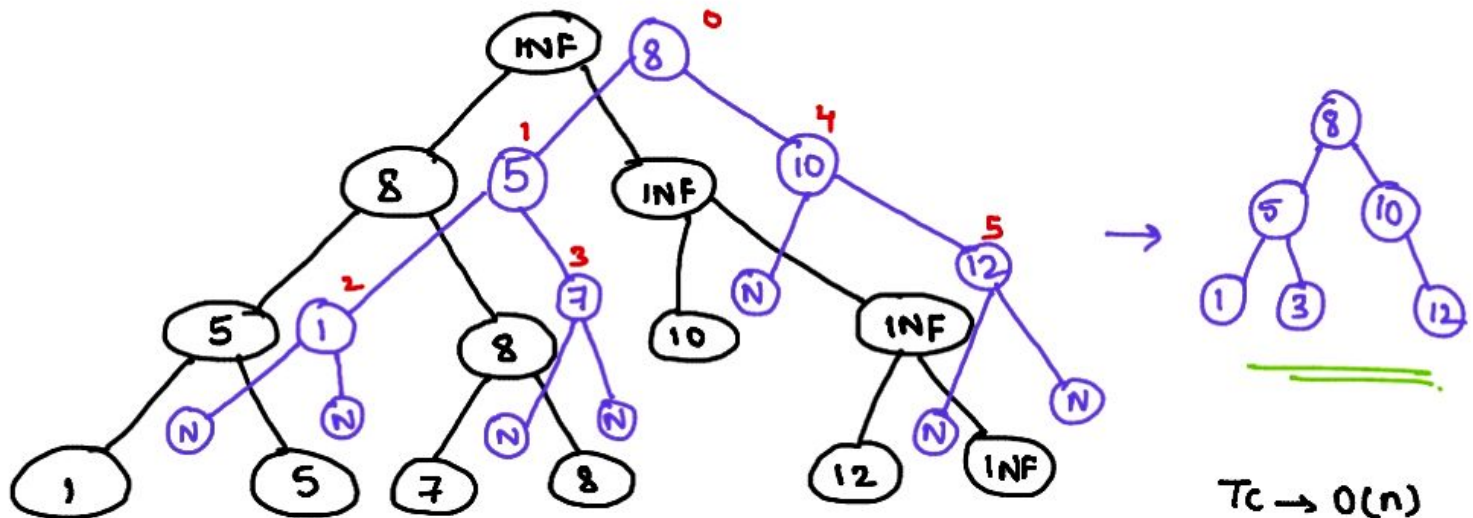
Approach 2  $\rightarrow$

[8, 5, 1, 7, 10, 12]

0 1 2 3 4 5

Boundary of LST  $\rightarrow$  Val  
RST  $\rightarrow$  boundVal

$\rightarrow$  initially (INF)



$T_c \rightarrow O(n)$   
 $S_c \rightarrow O(n)$

code  $\rightarrow$

```
1 class Solution {
2 public:
3     TreeNode* buildTree(vector<int>& preorder, int &preIdx, int boundary){
4         if(preIdx >= preorder.size() || preorder[preIdx] >= boundary)
5             return NULL;
6
7         // create root using preIdx
8         TreeNode* root = new TreeNode(preorder[preIdx]);
9         preIdx++;
10
11        // recursively call LST & RST
12        root->left = buildTree(preorder, preIdx, root->val);
13        root->right = buildTree(preorder, preIdx, boundary);
14        return root;
15    }
16
17    TreeNode* bstFromPreorder(vector<int>& preorder) {
18        int preIdx = 0;
19        return buildTree(preorder, preIdx, 1001);
20    }
21 };
22
```



# Graph

- Karun Karthik

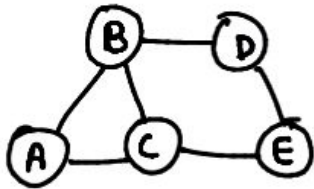
## Contents

0. Introduction
1. All paths from source to target
2. Flood Fill
3. Number of Islands
4. Max Area of the Island
5. Find if path exist in Graph
6. Find the town judge
7. Detect cycle in a Directed Graph
8. Topological Sort
9. Course Schedule
10. Course Schedule II

# Graphs

Graph  $G$  is a pair  $(V, E)$  where  $V$  is set of vertices &  $E$  is set of edges.  $n = |V|$  &  $e = |E|$

Eg



$$V = \{A, B, C, D, E\} \quad n = 5$$

$$E = \{AB, AC, BC, BD, CE, DE\} \quad e = 6$$

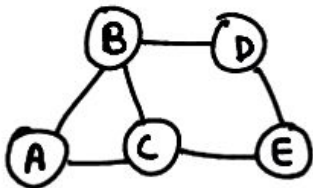
## Applications →

Google maps → To find shortest routes

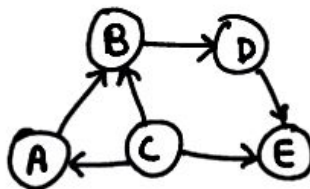
Social network → user, connection  
                                  ↑                  ↑  
                                 vertex      edge

## Types →

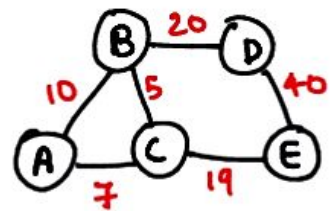
1) Undirected



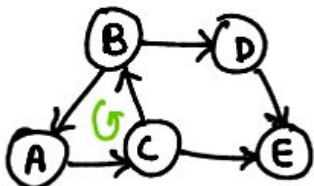
2) Directed



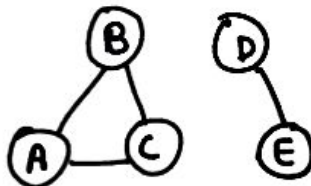
3) Weighted



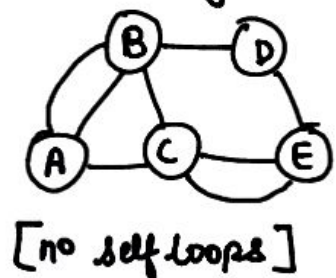
4) Cyclic



5) Disconnected



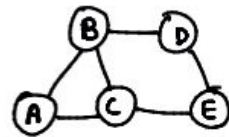
6) Multigraph



# Graph Traversal

(a) BFS → visit each and every vertex in a defined order.

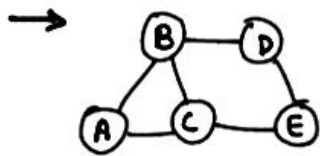
- select node
- visit its unvisited neighbour nodes
- mark it as visited & push into result
- push it into queue
- if no neighbours then pop.
- repeat till queue is empty



queue.

visited

A	B	C	D	E

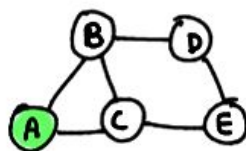


queue.

visited

A	B	C	D	E

res

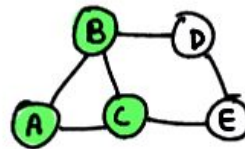


queue.

visited

A	B	C	D	E

res

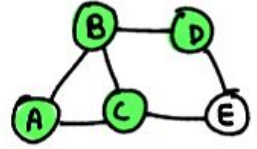


queue.

visited

A	B	C	D	E

res

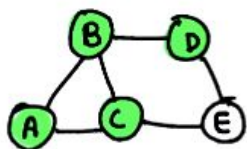


queue.

visited

A	B	C	D	E

res

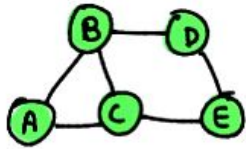


queue.

visited

A	B	C	D	E

res

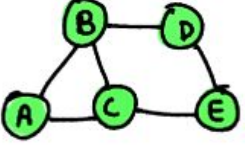


queue.

visited

A	B	C	D	E

res

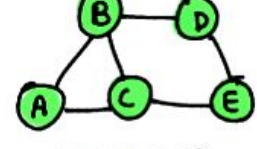


queue.

visited

A	B	C	D	E

res

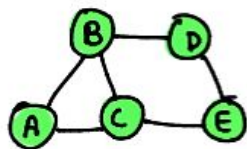


queue.

visited

A	B	C	D	E

res

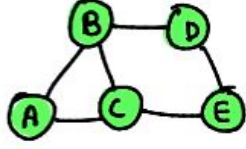


queue.

visited

A	B	C	D	E

res



queue.

visited

A	B	C	D	E

res

TC →  $O(V+E)$

SC →  $O(V)$

Return res

## Code

```
class Solution {
public:

    vector<int> bfsOfGraph(int V, vector<int> adj[]) {
        vector<int> ans;
        vector<int> vis(V, 0);
        queue<int> q;
        q.push(0);

        while(!q.empty()){
            int curr = q.front();
            q.pop();
            vis[curr]=1;
            ans.push_back(curr);
            for(auto it:adj[curr]){
                if(vis[it]==0){
                    vis[it]=1;
                    q.push(it);
                }
            }
        }
        return ans;
    }
};
```

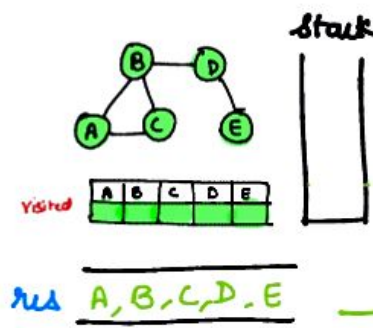
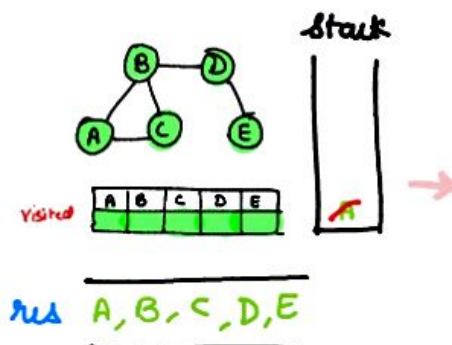
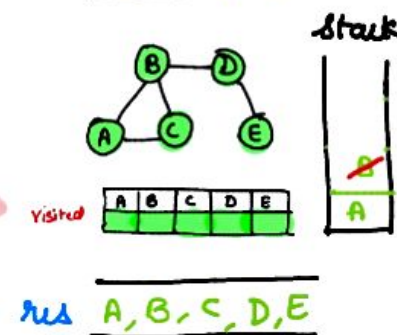
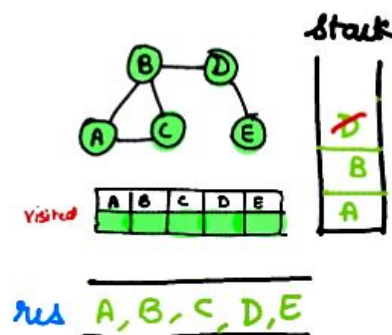
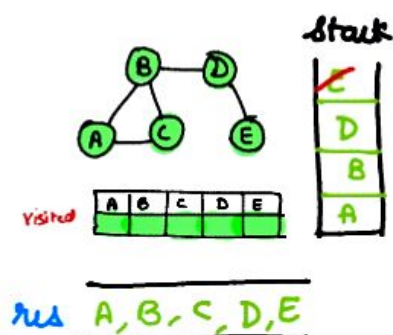
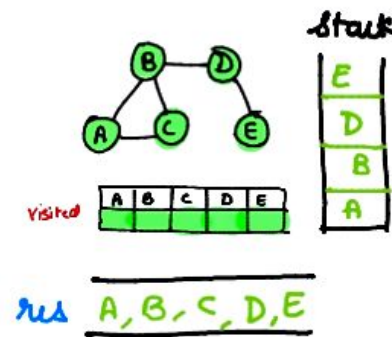
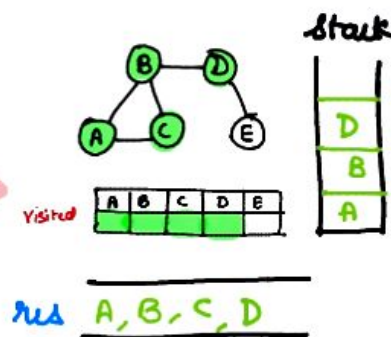
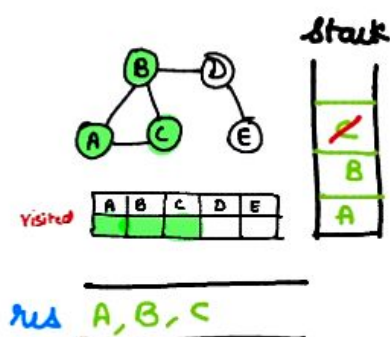
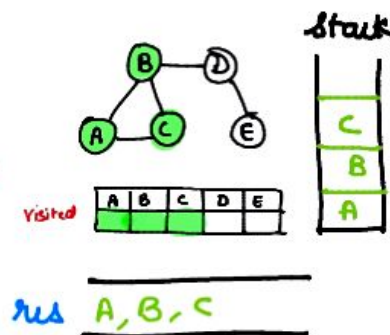
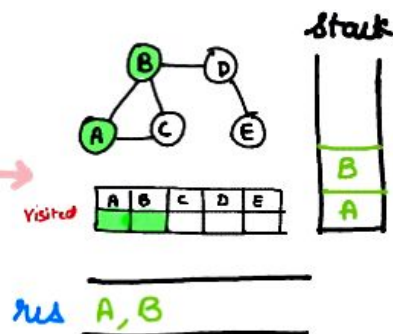
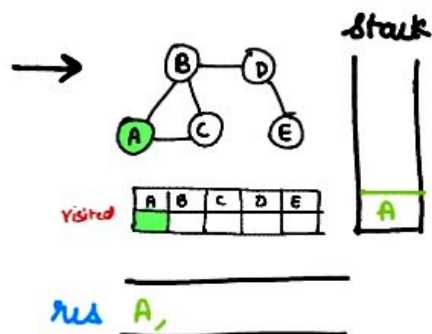
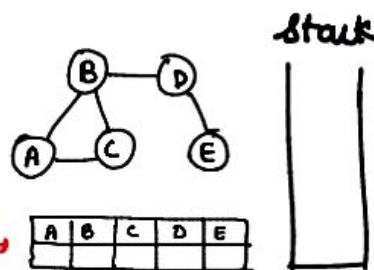
## Applications → [BFS]

1. Shortest path
2. Min. spanning tree for unweighted graph
3. Cycle detection
4. GPS
5. Social network.



# ⑥ DFS →

- select node
- visit its unvisited neighbour nodes
- mark it as visited & push into result
- push it into stack
- if no neighbours then pop.
- repeat till stack is empty



$$TC \rightarrow O(V+E)$$

$$SC \rightarrow O(V)$$

→ Return res.

## code

```
class Solution {
public:

    void dfs(vector<int>&ans, vector<int>&vis, int node, vector<int>adj[]){
        vis[node] = 1;
        ans.push_back(node);
        for(auto it:adj[node]){
            if(!vis[it]){
                vis[it] = 1;
                dfs(ans, vis, it, adj);
            }
        }
    }

    vector<int> dfsOfGraph(int V, vector<int> adj[]) {
        vector<int> ans;
        vector<int> vis(V,0);
        for(int i=0; i<V; i++){
            if(vis[i]==0)
                dfs(ans, vis, i, adj);
        }
        return ans;
    }
};
```

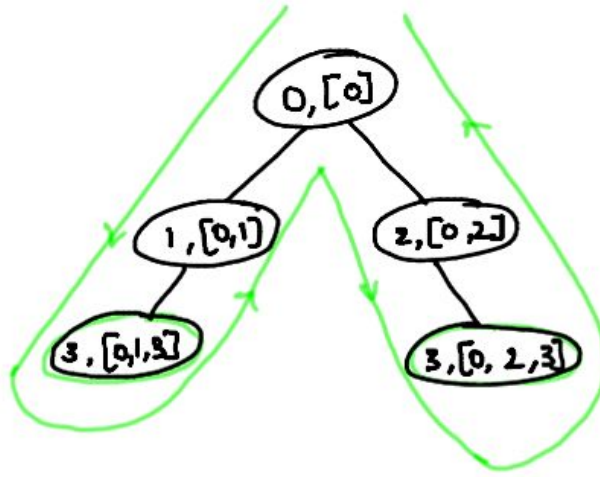
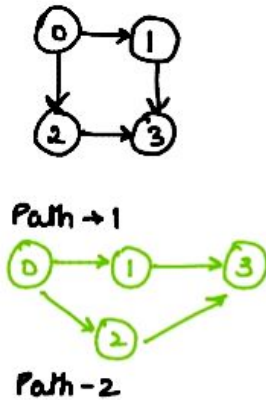
## Applications → [DFS]

1. Path finding
2. Cycle detection
3. Topological sort
4. Finding strongly connected components.

# ① All paths from src to target

Given a directed acyclic graph, return all paths from node 0 to node n-1.

Eg



Code →

TC →  $O(V+E)$

$V$  → Vertices

$E$  → edges

SC →

Recursive Stack

+ Results

```
1 class Solution {
2 public:
3     void findAllPaths(vector<vector<int>>&graph, int currNode, vector<bool>&visited,
4                       int n, vector<int> &currPath, vector<vector<int>>&res){
5
6         if(currNode==n-1){
7             res.push_back(currPath);
8             return;
9         }
10
11         if(visited[currNode]==true) return;
12
13         // backtrack for every node
14         visited[currNode] = true;
15
16         for(auto neighbour: graph[currNode]){
17             currPath.push_back(neighbour);
18             findAllPaths(graph, neighbour, visited, n, currPath, res);
19             currPath.pop_back();
20         }
21
22         visited[currNode] = false;
23     }
24
25     vector<vector<int>> allPathsSourceTarget(vector<vector<int>>& graph) {
26         vector<vector<int>> res;
27         vector<int> currPath;
28         int n = graph.size();
29         vector<bool> visited(n);
30
31         // traversing from 0 node
32         currPath.push_back(0);
33
34         findAllPaths(graph, 0, visited, n, currPath, res);
35         return res;
36     }
37 };
```



② Flood Fill → If 0 then blocker, else fill it with given color.

	0	1	2
0	1	1	1
1	1	1	0
2	1	0	1



2	2	2
2	2	0
2	0	1

\* perform flood fill from given index in all 4-directions & the cell should have same color as src

✓ let's follow the order to fill → UP, DOWN, LEFT, RIGHT

Eg In above case starting point is (1,1) & value = 1 so

④

1	1	1
1	1	0
1	0	1



1	1	1
1	2	0
1	0	1

up

1	2	1
1	2	0
1	0	1

← from here up is not possible & down is filled so left

left

2	2	1
1	2	0
1	0	1

down

2	2	1
2	2	0
1	0	1

down

2	2	1
2	2	0
2	0	1

→ no direction is possible so return

from here up not possible so down

from here up not possible so down

2	2	1
2	2	0
2	0	1

right

2	2	2
2	2	0
2	0	1

no other way possible

Result

2	2	2
2	2	0
2	0	1



Code

```
1  class Solution {
2  public:
3      void floodFiller(vector<vector<int>>& image, int i, int j,
4          int m, int n, int currColor, int newColor)
5      {
6          if(i<0 || i>=m || j<0 || j>= n || image[i][j] == newColor
7              || image[i][j] != currColor)
8              return;
9
10         image[i][j] = newColor;
11         floodFiller( image, i-1, j, m, n, currColor, newColor);
12         floodFiller( image, i+1, j, m, n, currColor, newColor);
13         floodFiller( image, i, j-1, m, n, currColor, newColor);
14         floodFiller( image, i, j+1, m, n, currColor, newColor);
15     }
16
17     vector<vector<int>> floodFill(vector<vector<int>>& image, int sr,
18         int sc, int newColor)
19     {
20         int m = image.size();
21         int n = image[0].size();
22         int currColor = image[sr][sc];
23         floodFiller(image, sr, sc, m, n, currColor, newColor);
24         return image;
25     }
26 };
```

$T_c \rightarrow O(mn)$

$S_c \rightarrow O(h)$

↳ recursive stack

③ Number of Islands → Given grid of 1 (land) & 0 (water), return no. of islands.

Eg

	0	1	2	3	4
0	[1, 1, 0, 0, 0],				
1	[1, 1, 0, 0, 0],				
2	[0, 0, 1, 0, 0],				
3	[0, 0, 0, 1, 1]				

- Always start dfs only if value = 1 & change its value to 0, so it cannot be visited again.
- if initial value = 0 then skip.
- initially ans = 0

• let's start from (0,0) & try moving U, D, L, R  
→ the traversal goes in this order

(0,0) → (1,0) → (1,1) → (0,1) is  
& update ans.

[1, 1, 0, 0, 0],
[1, 1, 0, 0, 0],
[0, 0, 1, 0, 0],
[0, 0, 0, 1, 1]

ans = 1.

→ now grid becomes

	0	1	2	3	4
0	[0, 0, 0, 0, 0],				
1	[0, 0, 0, 0, 0],				
2	[0, 0, 1, 0, 0],				
3	[0, 0, 0, 1, 1]				

• now, we can skip every entry from (1,0) to (2,1) as they are 0s.

• now start from (2,2), as U, D, L, R is not possible, set its value = 0 & update ans.  
ans = 2.

→ now grid becomes

	0	1	2	3	4
0	[0, 0, 0, 0, 0],				
1	[0, 0, 0, 0, 0],				
2	[0, 0, 0, 0, 0],				
3	[0, 0, 0, 1, 1]				

• now, we can skip every entry from (2,3) to (3,2) as they are 0s.

• now start from (3,3), it goes as follows  
(3,3) → (3,4)

• further traversal from (3,4) is not possible

ans = 3. ans = 3

## Code

```
1 class Solution {
2 public:
3     void countIsland(vector<vector<char>>& grid, int currRow, int currCol, int row, int col){
4         if(currRow<0 || currRow>=row || currCol<0 || currCol>=col || grid[currRow][currCol]=='0')
5             return;
6
7         grid[currRow][currCol] = '0';
8         countIsland(grid, currRow-1, currCol, row, col);
9         countIsland(grid, currRow+1, currCol, row, col);
10        countIsland(grid, currRow, currCol-1, row, col);
11        countIsland(grid, currRow, currCol+1, row, col);
12    }
13
14    int numIslands(vector<vector<char>>& grid) {
15        int ans = 0;
16        int row = grid.size();
17        int col = grid[0].size();
18
19        for(int currRow = 0; currRow < row; currRow++)
20            for(int currCol = 0; currCol < col; currCol++)
21                if(grid[currRow][currCol]=='1'){
22                    ans++;
23                    countIsland(grid, currRow, currCol, row, col);
24                }
25
26        return ans;
27    }
28 };
```

$T_c \rightarrow O(mn)$  Avg case  
 $O(m^2n^2)$  Worst case

#### ④ Max Area of the Island

- \* Intuition is same as previous problem.
- \* Minor Tweak to count number of 1s in island.
- \* Once entire island Traversal is done,  
compare for max Area of island.

T.C.  $\rightarrow O(mn)$  Avg case.

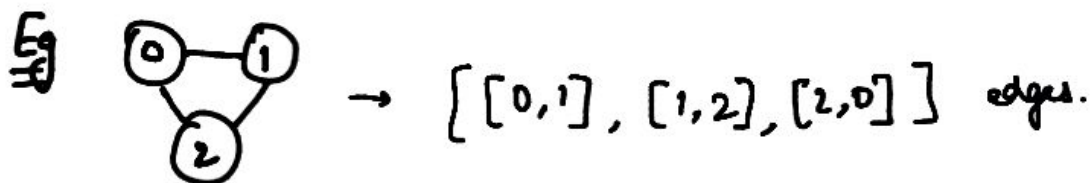
code  $\rightarrow$

```
1 class Solution {
2 public:
3     int findArea(vector<vector<int>>& grid, int currRow, int currCol, int m, int n){
4         if(currRow<0 || currCol<0 || currRow>=m || currCol>=n || grid[currRow][currCol]==0)
5             return 0;
6
7         grid[currRow][currCol]=0;
8
9         // this is for single cell where we started traversing
10        int count = 1;
11        count += findArea(grid, currRow-1, currCol, m, n);
12        count += findArea(grid, currRow+1, currCol, m, n);
13        count += findArea(grid, currRow, currCol-1, m, n);
14        count += findArea(grid, currRow, currCol+1, m, n);
15        return count;
16    }
17    int maxAreaOfIsland(vector<vector<int>>& grid) {
18        int m = grid.size();
19        int n = grid[0].size();
20        int ans = 0;
21        for(int currRow = 0; currRow<m; currRow++){
22            for(int currCol = 0; currCol<n; currCol++){
23                if(grid[currRow][currCol]==1){
24                    ans = max(ans, findArea(grid, currRow, currCol, m, n));
25                }
26            }
27        }
28        return ans;
29    };
}
```



⑤ Find if path exist in graph.

Given src, dest, no. of nodes & set of edges, find if path exist b/w src & dest.

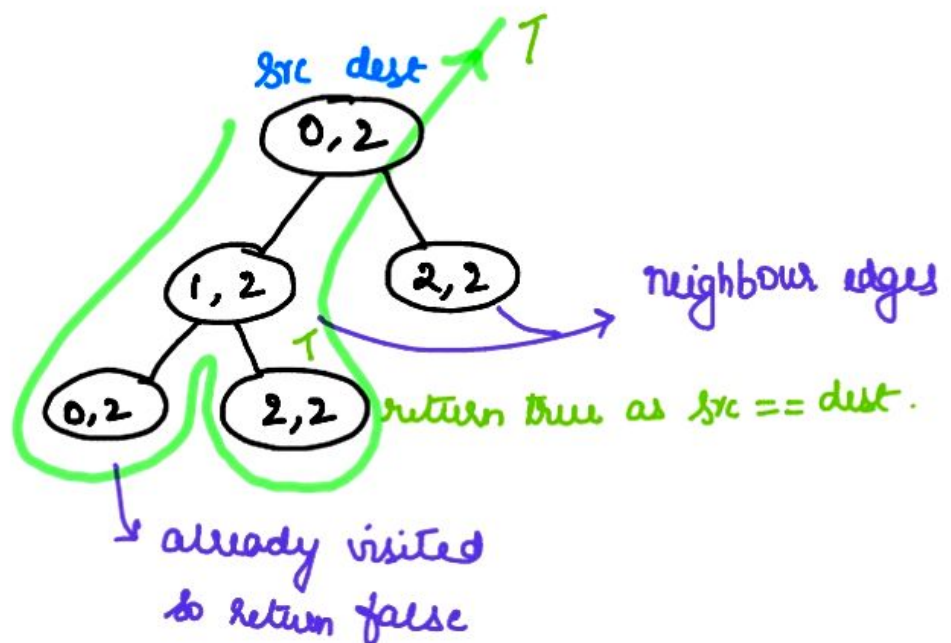


$n=3$  edges =  $[[0,1], [1,2], [2,0]]$  src = 0, dest = 2.

- 1) Create a graph using adj list rep.  $[[1,2], [0,2], [1,0]]$   
0 1 2
- 2) Perform dfs

$[[1,2], [0,2], [1,0]]$   
0 1 2

<del>T</del>	<del>F</del>	F
0	1	2

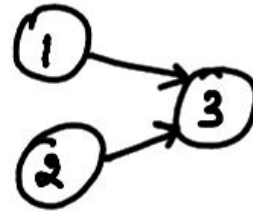


Code →

```
1  class Solution {
2  public:
3      bool validPath(int n, vector<vector<int>>& edges, int src, int dest) {
4
5          vector<vector<int>>graph(n);
6          for(int i=0;i<edges.size();i++)
7          {
8              int v1 = edges[i][0];
9              int v2 = edges[i][1];
10             graph[v1].push_back(v2);
11             graph[v2].push_back(v1);
12
13         }
14         vector<bool>vis(n,false);
15         return pathExist(src, dest, graph, vis);
16     }
17
18     bool pathExist(int src , int dest,vector<vector<int>>&graph,vector<bool>&vis){
19
20         if(src==dest)return true;
21
22         vis[src]=true;
23
24         for(int i=0;i<graph[src].size();i++)
25             if(vis[graph[src][i]]==false)
26                 if(pathExist(graph[src][i],dest,graph,vis)==true)
27                     return true;
28
29         return false;
30     }
31 };
```

⑥ Find the town judge

$n = 3$ ,  $trust = [[1, 3], [2, 3]]$  →



\* Indegree of town judge =  $n-1$   
& outdegree = 0.

✓ Create 2 arrays

outdegree	<table><tr><td><del>0</del> 1</td><td>0</td><td><del>0</del> 1</td><td>0</td></tr><tr><td>0</td><td>1</td><td>2</td><td>3</td></tr></table>	<del>0</del> 1	0	<del>0</del> 1	0	0	1	2	3
<del>0</del> 1	0	<del>0</del> 1	0						
0	1	2	3						
indegree	<table><tr><td>0</td><td>0</td><td>0</td><td><del>0</del> 2</td></tr><tr><td>0</td><td>1</td><td>2</td><td>3</td></tr></table>	0	0	0	<del>0</del> 2	0	1	2	3
0	0	0	<del>0</del> 2						
0	1	2	3						

for  $[1, 3]$

indegree of 1 ↑

outdegree of 3 ↑

for  $[2, 3]$

indegree of 2 ↑

outdegree of 3 ↑

→ traverse both indegree & outdegree

if indegree == 0 &&

outdegree ==  $n-1$

then return that vertex

code

```
1  class Solution {
2  public:
3      int findJudge(int n, vector<vector<int>>& trust) {
4          vector<int> indegree(n+1,0);
5          vector<int> outdegree(n+1,0);
6          for(int i=0;i<trust.size();i++)
7          {
8              int v1 = trust[i][0];
9              int v2 = trust[i][1];
10             outdegree[v1]+=1;
11             indegree[v2]+=1;
12         }
13         for(int i=1;i<=n;i++)
14         {
15             if(outdegree[i]==0 && indegree[i]==n-1)
16                 return i;
17         }
18         return -1;
19     }
20 };
```

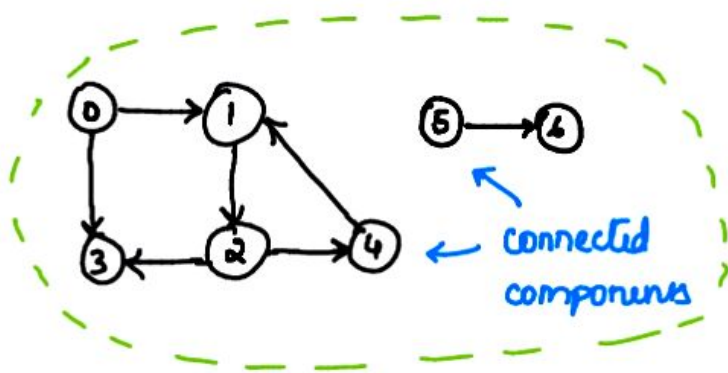


⑦ Detect cycle in a directed graph

Consider a graph with 'n' vertices labelled as  $[0..n-1]$

Ex  $n = 7$   $[0, 1, 2, 3, 4, 5, 6]$

Graph  $\rightarrow$



\* To detect cycle, check for backedge.

Let's start dfs from 0 vertex.

\* At every vertex, check if its already visited, if already visited then check if it is present in recursive track.

If present, then it indicates back edge  $\rightarrow$  Returns True

\* If vertex is not visited then mark it in visited array & recursive stack

visited  $\rightarrow \{0, 1, 2, 3, 4\}$

Recursive stack  $\rightarrow \{0, 1, 2, \cancel{3}, 4\}$

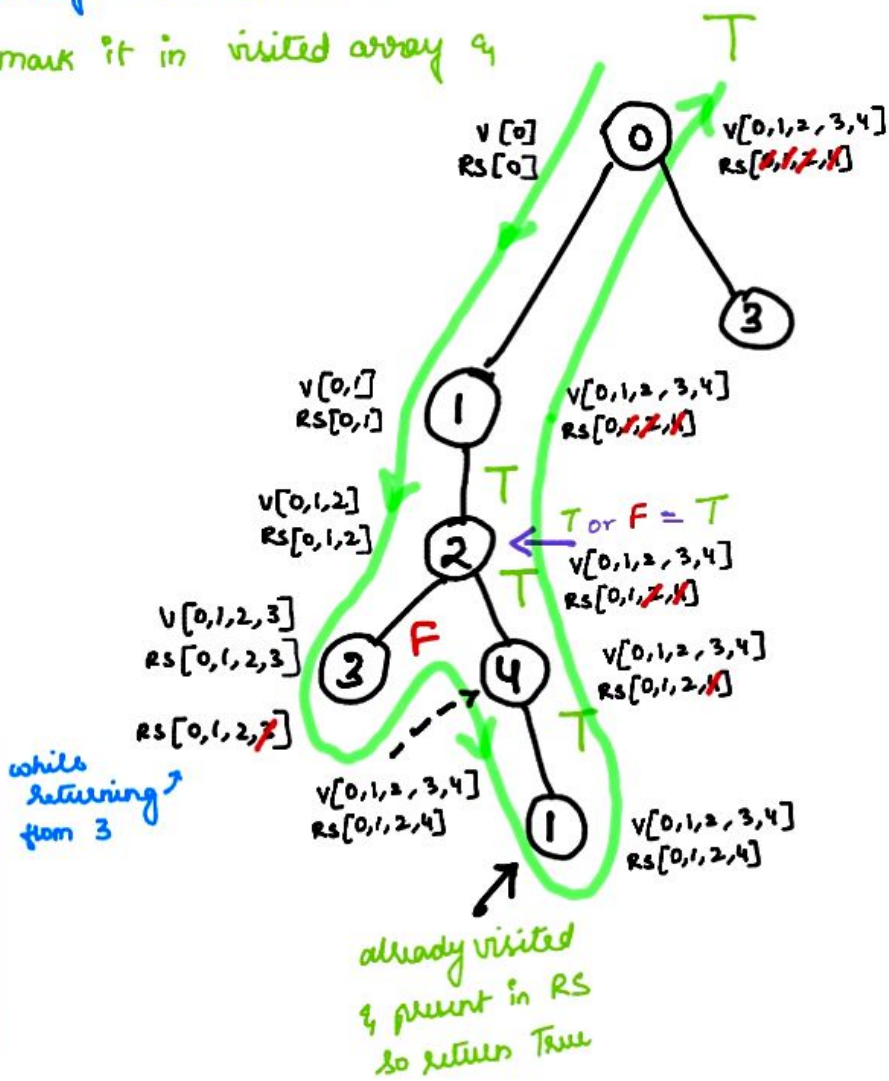
\* At 3 vertex, there's no neighbour  
 & no cycle is detected so return F.

Before returning, undo change made in Recursive stack by popping it.

visited  $\rightarrow \{0, 1, 2, 3, 4, 1\}$  6

Recursive stack  $\rightarrow \{0, 1, 2, \cancel{3}, 4\}$

1 is already present in recursive stack so return true.



Code

$T_c \rightarrow O(V+E)$

$S_c \rightarrow O(V)$

```
1  class Solution {
2      public:
3          bool dfs(int node, vector<int>&vis, vector<int>&rs, vector<int> adj[])
4          {
5              vis[node]=1;
6              rs[node]=1;
7              for(auto it:adj[node])
8              {
9                  if(vis[it]==0){
10                     if(dfs(it,vis,rs,adj))
11                         return true;
12                 }
13                 else if(rs[it]==1)
14                     return true;
15             }
16             rs[node]=0;
17             return false;
18         }
19         bool isCyclic(int V, vector<int> adj[]) {
20
21             vector<int>vis(V,0);
22             vector<int>rs(V,0);
23
24             for(int i=0;i<V;i++)
25             {
26                 if(vis[i]==0)
27                     if(dfs(i,vis,rs,adj))
28                         return true;
29             }
30             return false;
31         }
32     };
```

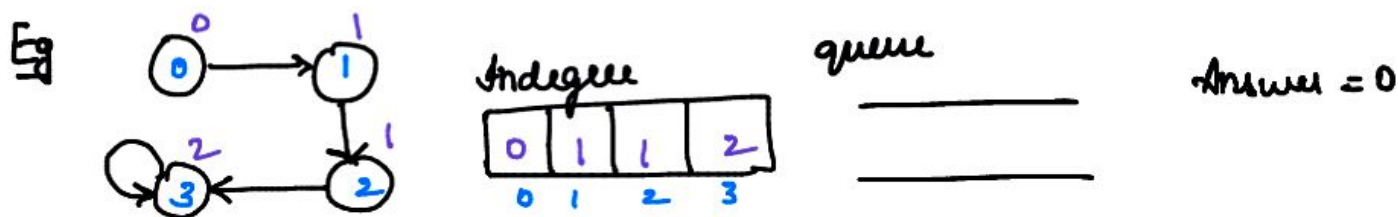
# ★ Kahn's Algorithm → To find topological Ordering

↓  
can be used to find cycle using BFS.

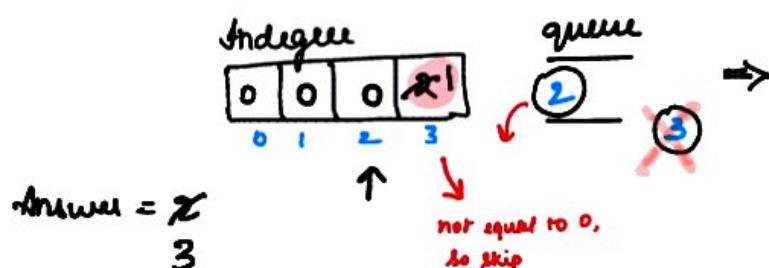
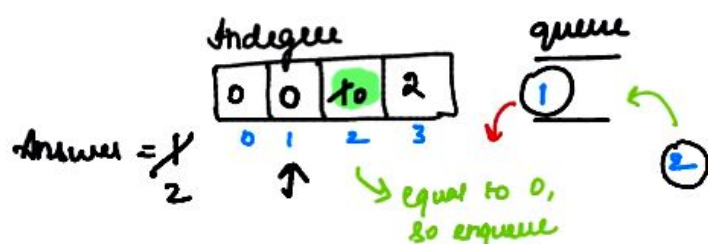
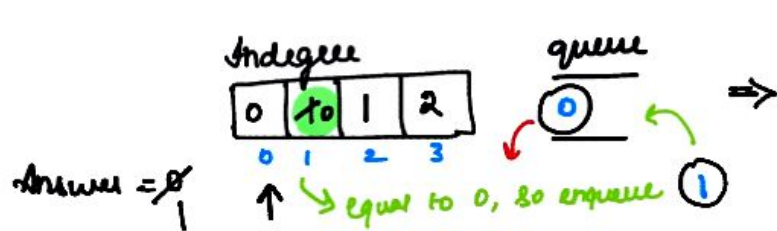
① Find indegree of every vertex in graph & answer = 0

② If indegree of vertex is 0, then push into queue & do bfs till queue is not empty & while doing bfs decrease the indegree of neighbours by 1.  
if indegree of neighbour = 0, then enqueue & increment answer by 1

③ If answer  $\neq$  no. of vertices then **cycle is present.**



→ As indegree of 0 is 0, we push into queue & do bfs till queue is not empty.



Answer = 3  
No of vertices = 4  
∴ cycle present.

code

```
1  class Solution{
2      public:
3          bool isCyclic(int V, vector<int> adj[]) {
4
5              vector<int> indegree(V,0);
6              for (int i = 0; i < V; i++)
7                  for(int it : adj[i])
8                      indegree[it]++;
9
10             queue<int> q;
11             int ans = 0;
12             unordered_set<int> vis;
13
14             for (int i=0; i<V; i++)
15             {
16                 if(indegree[i]==0){
17                     q.push(i);
18                     ans+=1;
19                 }
20             }
21
22             while(!q.empty())
23             {
24                 int currvertex = q.front();
25                 q.pop();
26                 if(vis.find(currvertex)!=vis.end())
27                     continue;
28                 vis.insert(currvertex);
29                 for(int neighbour:adj[currvertex])
30                 {
31                     indegree[neighbour]-=1;
32                     if(indegree[neighbour]==0)
33                     {
34                         q.push(neighbour);
35                         ans+=1;
36                     }
37                 }
38             }
39             if(ans==V) return false;
40             return true;
41         }
42     };
```



## ⑧ Topological sort

→ use Kahn's algorithm. & add node to result while performing dfs.

Code →

$TC \rightarrow O(V + E)$

$SC \rightarrow O(V)$

```
1  class Solution
2  {
3      public:
4      vector<int> topoSort(int V, vector<int> adj[])
5      {
6          vector<int> indegree(V,0) ,res;
7
8          for(int i=0; i<V; i++)
9              for(auto it:adj[i])
10                 indegree[it]++;
11
12         queue<int> q;
13         int ans = 0;
14         unordered_set<int> vis;
15
16         for(int i=0; i<V; i++)
17         {
18             if(indegree[i]==0){
19                 q.push(i);
20                 ans+=1;
21             }
22         }
23
24         while(!q.empty())
25         {
26             int curr = q.front();
27             q.pop();
28
29             // add to res
30             res.push_back(curr);
31
32             if(vis.find(curr)!=vis.end())
33                 continue;
34
35             vis.insert(curr);
36
37             for(int neighbour: adj[curr])
38             {
39                 indegree[neighbour]-=1;
40                 if(indegree[neighbour]==0)
41                 {
42                     q.push(neighbour);
43                     ans+=1;
44                 }
45             }
46         }
47
48         return res;
49     }
50 };
```

⑨ Course Schedule → can be solved using Kahn's algo.

$$Tc \rightarrow O(V+E)$$

$$Sc \rightarrow O(V+E)$$

code →

```
1 class Solution {
2 public:
3     vector<vector<int>> createGraph(int n, vector<vector<int>>& pre){
4         vector<vector<int>> graph(n);
5         for(auto it:pre){
6             int v = it[1];
7             int u = it[0];
8             graph[v].push_back(u);
9         }
10        return graph;
11    }
12
13    bool canFinish(int n, vector<vector<int>>& pre) {
14        vector<vector<int>> graph = createGraph(n, pre);
15        vector<int> indegree(n,0);
16        for(int i=0; i<n; i++)
17            for(int it: graph[i])
18                indegree[it]++;
19
20        queue<int> q;
21        int ans = 0;
22        unordered_set<int> vis;
23
24        for(int i=0; i<n; i++)
25            if(indegree[i]==0){
26                q.push(i);
27                ans++;
28            }
29
30        while(!q.empty()){
31            int currvertex = q.front();
32            q.pop();
33            if(vis.find(currvertex)!=vis.end())
34                continue;
35            vis.insert(currvertex);
36            for(int neighbour: graph[currvertex]){
37                indegree[neighbour]--;
38                if(indegree[neighbour]==0){
39                    q.push(neighbour);
40                    ans++;
41                }
42            }
43        }
44        if(ans==n) return true;
45        return false;
46    }
47 };
```

# ⑩ Course Schedule - II

$n \rightarrow$  no. of courses [vertices]

Topological sort only for DAG.

Eg  $n \rightarrow 4$  (0, 1, 2, 3)

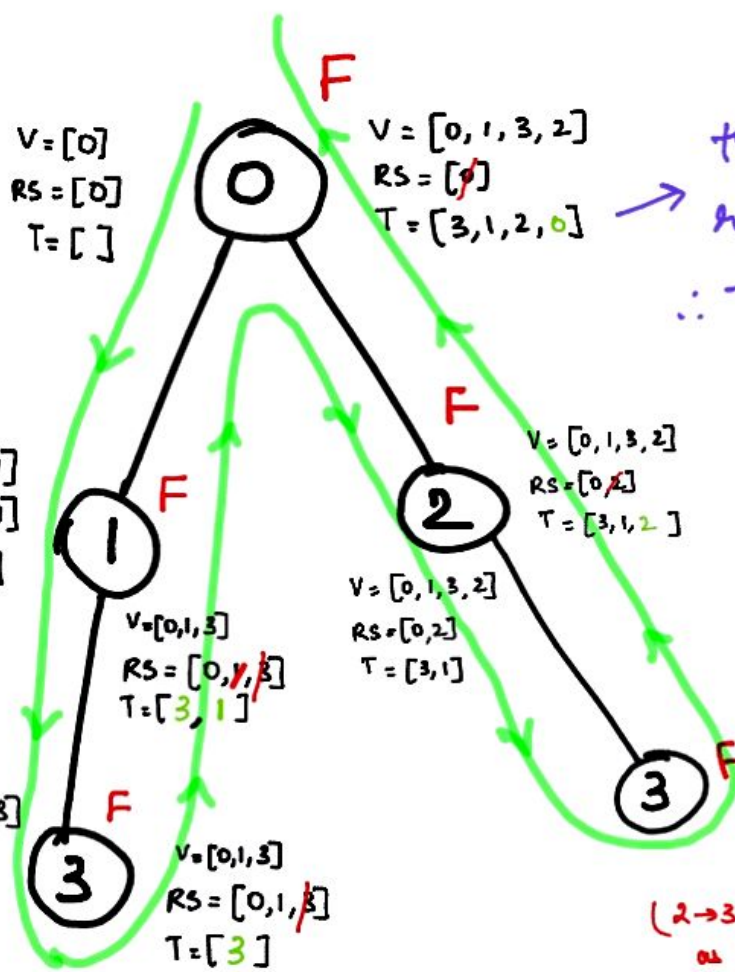
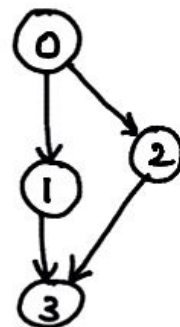
$pre \rightarrow [[1, 0], [2, 0], [3, 1], [3, 2]]$

Initially

$V = [ ]$ ,  $RS = [ ]$ ,  $traversal = [ ]$

$pre \rightarrow$  edges  $[v, u]$

"u should be completed before v"



topological order in reverse!

$\therefore T = [0, 1, 2, 3]$

(2  $\rightarrow$  3 is not backedge as it is not in RS)

while returning from 3  $\uparrow$   
 pop 3 & push into traversal array.  
 return F, as no cycle is found

code →

$$T_c \rightarrow O(V+E)$$

$$S_c \rightarrow O(V+E)$$

```
1  class Solution {
2  public:
3      bool dfs(vector<vector<int>>&graph, int i, vector<int> &vis,
4              vector<int> &rs, vector<int> &traversal){
5
6          vis[i] = 1;
7          rs[i] = 1;
8          for(int neighbour: graph[i]){
9              if(vis[neighbour]==0){
10                 if(dfs(graph, neighbour, vis, rs, traversal))
11                     return true;
12             }
13             else if(rs[neighbour]==1) return true;
14         }
15         traversal.push_back(i);
16         rs[i]=0;
17         return false;
18     }
19
20     vector<vector<int>> createGraph(int n, vector<vector<int>>& pre){
21         vector<vector<int>> graph(n);
22         for(auto it:pre){
23             int v = it[1];
24             int u = it[0];
25             graph[v].push_back(u);
26         }
27         return graph;
28     }
29
30     vector<int> findOrder(int n, vector<vector<int>>& pre) {
31         vector<vector<int>> graph = createGraph(n, pre);
32         vector<int> vis(n,0), rs(n,0), traversal;
33         for(int i=0; i<n; i++){
34             if(vis[i]==0)
35                 if(dfs(graph, i, vis, rs, traversal)) return {};
36         }
37         reverse(traversal.begin(), traversal.end());
38         return traversal;
39     }
40 };
```



# Graph - 2

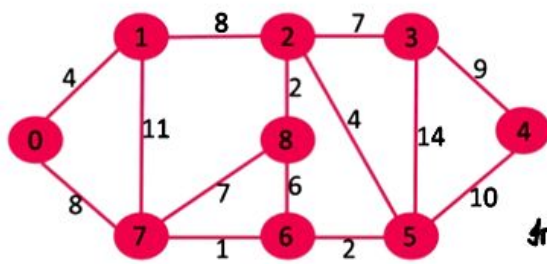
- Karun Karthik

## Content

11. Dijkstra Algorithm
12. Network Delay Time
13. Bellman Ford Algorithm
14. Negative Weight Cycle
15. Floyd Warshall Algorithm
16. Prim's Algorithm
17. Min Cost to Connect All Points
18. Is Graph Bipartite ?
19. Possible Bipartition
20. Disjoint Set
21. Kruskal's Algorithm
22. Critical Connection in a Network

# ⑪ Dijkstra Algorithm → single source shortest path (only +ve weights)

→ Helps in finding the shortest path to every node from src node.



$n = 9$  (nodes from 0 to 8)

src = ①

cost away = min cost from src to every other vertex

Initially cost = 

0	0	0	0	0	0	0	0	0
0	1	2	3	4	5	6	7	8

 vis = { 3 }

→ As it is weighted graph, we'll use priority queue (pq) instead of normal queue. & element pushed into it will be of form curr node, curr cost

→ pq always pops element with least curr cost. always calculated from src to curr node.

✓ ⇒ now neighbours of 1 = 0, 4 7, 11 2, 8 ∴ push



vis = {1, 3}

cost[1] = 0



→ lowest cost among 4, 11, 8 is 4 ∴ pop it & push its neighbours.

⇒ 1, 0 0, 4 7, 11 2, 8 ⇒ now neighbours of 0 = 1 (visited), 7, 12 ∴ push

vis = {1, 0, 3}

cost[0] = 4



→ lowest cost is 8 ∴ pop & push its neighbours

⇒ 1, 0 0, 4 7, 11 2, 8 7, 12 ⇒ neighbours of 2 = 1 (visited), 8, 10, 3, 15, 5, 12 ∴ push

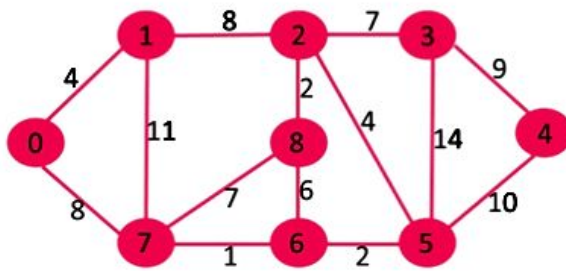
vis = {1, 0, 2, 3}

cost[2] = 8



→ lowest cost is 10 ∴ pop & push its neighbours





⇒

~~(1,0)~~ ~~(0,4)~~ ~~(7,11)~~ ~~(2,8)~~ ~~(7,12)~~ **(8,10)** ~~(3,15)~~ ~~(5,12)~~ ⇒ neighbours of 8 = 2 (visited), **(7,17)**, **(6,16)** ∴ push

vis = {1, 0, 2, 8}

cost[8] = 10

~~(1,0)~~ ~~(0,4)~~ ~~(7,11)~~ ~~(2,8)~~ ~~(7,12)~~ ~~(8,10)~~ ~~(3,15)~~ ~~(5,12)~~ **(7,17)** **(6,16)**

↳ lowest cost = 11 ∴ pop & push its neighbours.

⇒

~~(1,0)~~ ~~(0,4)~~ **(7,11)** ~~(2,8)~~ ~~(7,12)~~ ~~(8,10)~~ ~~(3,15)~~ ~~(5,12)~~ ~~(7,17)~~ ~~(6,16)~~ ⇒ neighbours of 7 = 0, 1, 8 are visited. & **(6,12)** ∴ push

vis = {1, 0, 2, 8, 7}

cost[7] = 11

~~(1,0)~~ ~~(0,4)~~ ~~(7,11)~~ ~~(2,8)~~ ~~(7,12)~~ ~~(8,10)~~ ~~(3,15)~~ ~~(5,12)~~ ~~(7,17)~~ ~~(6,16)~~ **(6,12)** ≈ **(7,17)** **(3,15)** **(5,12)** **(7,17)** **(6,16)** **(6,12)**

↳ lowest cost = 12

∴ Anything among 5, 6 can be selected & pop & push its neighbours  
Not 7, because its already visited & cost is < 12.

⇒

**(3,15)** **(5,12)** ~~(7,17)~~ ~~(6,16)~~ ~~(6,12)~~ ⇒ neighbours of 5 = **(4,22)**, **(3,24)**, **(6,14)** ∴ push

vis = {1, 0, 2, 8, 7, 5}

cost[5] = 12

~~(3,15)~~ ~~(5,12)~~ ~~(7,17)~~ ~~(6,16)~~ **(6,12)** **(4,22)** **(3,24)** **(6,14)**

→ lowest cost = 12  
∴ pop & push its neighbours

⇒

~~(3,15)~~ ~~(7,17)~~ ~~(6,16)~~ **(6,12)** **(4,22)** **(3,24)** **(6,14)**

⇒ neighbours of 6 = 5, 7, 8 are visited.  
∴ no push

vis = {1, 0, 2, 8, 7, 5, 6}

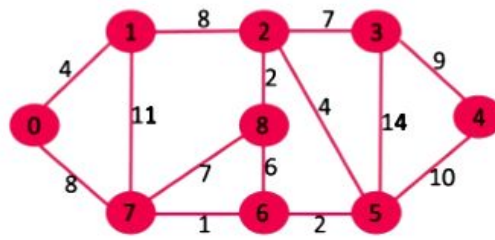
cost[6] = 12

~~(3,15)~~ ~~(7,17)~~ ~~(6,16)~~ ~~(6,12)~~ ~~(4,22)~~ ~~(3,24)~~ **(6,14)**

→ next lowest is 14, but 6 is already visited.

**(3,15)** **(7,17)** **(6,16)** **(4,22)** **(3,24)** ~~(6,14)~~

∴ Next lowest is 15, ∴ pop & push its neighbours



⇒ (3, 15), (7, 17), (6, 16), (4, 22), (3, 26) ⇒ neighbours of 3 = 2, 5 (visited), (4, 24) ∴ push

vis = {1, 0, 2, 8, 7, 5, 6, 3}  
cost[3] = 15

~~(3, 15)~~, (7, 17), (6, 16), (4, 22), (3, 26), (4, 24) → next lowest cost = 16 but 6 is already visited ∴ pop

(7, 17), ~~(6, 16)~~, (4, 22), (3, 26), (4, 24) ⇒ (4, 22), (3, 26), (4, 24) → next lowest cost = 22  
∴ pop & push its neighbours  
→ next lowest cost = 17 but 7 is already visited ∴ pop

⇒ (4, 22), (3, 26), (4, 24) ⇒ neighbours of 4 = 3, 5 (visited) ∴ no push

vis = {1, 0, 2, 8, 7, 5, 4, 3, 4}  
cost[4] = 22

~~(4, 22)~~, (3, 26), (4, 24) → next lowest cost = 24 but 4 is already visited ∴ pop

(3, 26), ~~(4, 24)~~ → next lowest cost = 26 but 3 is already visited ∴ pop  
⇒ ~~(3, 26)~~ ⇒ \_\_\_\_\_ ∴ empty PQ.

Answer ⇒

cost = 

4	0	8	15	22	12	12	11	10
0	1	2	3	4	5	6	7	8

Dijkstra's = BFS + PQ

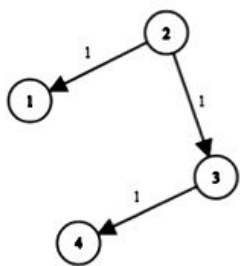
Tc →  $O(V + E \log V)$   
Sc →  $O(V)$



Code →

```
1  class Solution
2  {
3      public:
4      vector<int> dijkstra(int V, vector<vector<int>> adj[], int src)
5      {
6          vector<int> cost(V, 0);
7          cost[src] = 0;
8
9          vector<bool> vis(V, false);
10         priority_queue<pair<int, int>, vector<pair<int, int>>, greater<pair<int, int>>> pq;
11
12         pq.push({0, src}); // {cost, node}
13
14         while(!pq.empty())
15         {
16             pair<int, int> p = pq.top();
17             int currCost = p.first;
18             int currNode = p.second;
19             pq.pop();
20
21             if(vis[currNode]) continue;
22
23             vis[currNode] = true;
24             cost[currNode] = currCost;
25
26             for(int i=0; i<adj[currNode].size(); i++)
27             {
28                 int neighbourNode = adj[currNode][i][0];
29                 int weight = adj[currNode][i][1];
30                 // if already visited then skip
31                 if(vis[neighbourNode]) continue;
32                 // else push
33                 pq.push({currCost + weight, neighbourNode});
34             }
35         }
36         return cost;
37     }
38 };
39
```

## (12) Network Delay Time



src = 2.

You are given a network of  $n$  nodes, labeled from 1 to  $n$ . You are also given times, a list of travel times as directed edges  $times[i] = (u_i, v_i, w_i)$ , where  $u_i$  is the source node,  $v_i$  is the target node, and  $w_i$  is the time it takes for a signal to travel from source to target.

We will send a signal from a given node  $k$ . Return the time it takes for all the  $n$  nodes to receive the signal. If it is impossible for all the  $n$  nodes to receive the signal, return  $-1$ .

∴ similar to dijkstra's algo.  $cost = \begin{bmatrix} 0 & 0 & 0 & 0 & 0 \end{bmatrix}$   $vis = \{ \}$   $pq = \underline{\hspace{2cm}}$

⇒ push  $(2, 0)$  to  $pq$ . ⇒  $\underline{(2, 0)}$

⇒  $\underline{(2, 0)}$  neighbours =  $(1, 1), (3, 1)$  ∴ push  
 $vis = \{2\}$   $cost[2] = 0$   
 $\underline{(2, 0), (1, 1), (3, 1)}$  → next lowest cost = 1 ∴ choose 1 or 3 pop & push their neighbours.

⇒  $\underline{(1, 1), (3, 1)}$  no new neighbours ∴ pop  
 $vis = \{2, 1\}$   $cost[1] = 1$   
 $\underline{(1, 1), (3, 1)}$  → next lowest cost = 1 ∴ pop & push their neighbours.

⇒  $\underline{(3, 1)}$  neighbour =  $(4, 2)$  ∴ push  
 $vis = \{2, 1, 3\}$   $cost[3] = 1$   
 $\underline{(3, 1), (4, 2)}$  → next lowest cost = 2 ∴ pop & push neighbours.

⇒  $\underline{(4, 2)}$  no new neighbours ∴ pop  
 $vis = \{2, 1, 3, 4\}$   $cost[4] = 2$   
 $\underline{(4, 2)}$  →  $pq$  is empty.

∴  $cost = \begin{bmatrix} 0 & 1 & 0 & 1 & 2 \end{bmatrix}$   
 $0 \ 1 \ 2 \ 3 \ 4$

→ check if all nodes are in visited, else return  $-1$ .

→ Return max value in cost as

$Tc \rightarrow O(V + E \log V)$   
 $Sc \rightarrow O(V)$

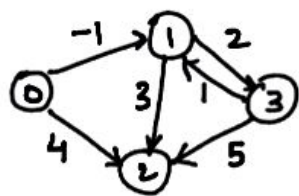
Code →

```
1 class Solution {
2 public:
3
4     int networkDelayTime(vector<vector<int>>& times, int n, int k) {
5         vector<vector<vector<int>>> graph = createGraph(times,n);
6         return minTime(graph,n,k);
7     }
8
9     vector<vector<vector<int>>> createGraph(vector<vector<int>>& edges,int n) {
10
11         vector<vector<vector<int>>> graph(n+1);
12
13         for(int i=0;i<=n;i++) {
14             graph.push_back({});
15         }
16         // add every edge to the graph
17         for(vector<int> edge:edges) {
18             int source = edge[0];
19             int dest = edge[1];
20             int cost = edge[2];
21             graph[source].push_back({dest,cost});
22         }
23         return graph;
24     }
25
26     int minTime(vector<vector<vector<int>>> &graph,int n,int src) {
27
28         vector<int> cost(n+1,0);
29         cost[src] = 0;
30         vector<bool> vis(n+1, false);
31
32         priority_queue<pair<int,int>,vector<pair<int,int>>,greater<pair<int,int>>> pq;
33         pq.push({0,src}); // {cost, node}
34
35         while(!pq.empty()) {
36             pair<int,int> p = pq.top();
37             int currNode = p.second;
38             int currCost = p.first;
39             pq.pop();
40             // if already visited then skip
41             if(vis[currNode]) continue;
42
43             vis[currNode] = true;
44             cost[currNode] = currCost;
45
46             for(int i=0;i<graph[currNode].size();i++)
47             {
48                 int neighbourNode = graph[currNode][i][0];
49                 int weight = graph[currNode][i][1];
50                 // if already visited then skip
51                 if(vis[neighbourNode]) continue;
52                 // else push into pq
53                 pq.push({currCost + weight, neighbourNode});
54             }
55         }
56
57         for(int i=1; i<=n; i++)
58             if(vis[i]==0) return -1;
59
60         int ans = 0;
61         for(int x:cost) ans = max(ans,x);
62         return ans;
63     }
64 };
```



(13) Bellman Ford Algorithm → useful when weights  $< 0$  (Dijkstra fails)  
 ↳ dp algo → useful when finding negative weight cycle.

Eg  $n = 4$  edges =  $\begin{matrix} [src, dest, wt] \\ [0, 1, -1], [0, 2, 4], [1, 2, 3], [1, 3, 2], [3, 1, 1], [3, 2, 5] \end{matrix}$



initially dist 

inf	inf	inf	inf
0	1	2	3

⇒  $dist[src] = 0$  &

⇒ relax every edge  $n-1$  time is run for loop & perform the following operation

$$dist[dest] = \min(dist[src] + weight, dist[dest])$$

⇒ finally relax one more time &

if  $dist[dest] > dist[src] + wt \Rightarrow$  -ve weight cycle present

⇒ we should relax 3 times &  $src = 0 \Rightarrow dist[src] = 0$  dist 

0	inf	inf	inf
0	1	2	3

→ for edge  $[0, 1, -1]$ ,  $dist[1] = \min(0 + (-1), inf) = -1$

$[0, 2, 4]$ ,  $dist[2] = \min(0 + 4, inf) = 4$

$[1, 2, 3]$ ,  $dist[2] = \min(-1 + 3, 4) = 2$

$[1, 3, 2]$ ,  $dist[3] = \min(-1 + 2, inf) = 1$

$[3, 1, 1]$ ,  $dist[1] = \min(1 + 1, -1) = -1$

$[3, 2, 5]$ ,  $dist[2] = \min(1 + 5, 2) = 2$ .

∴ dist = 

0	-1	2	1
0	1	2	3

→ now use the above dist & perform same operation twice, in this case dist remains same.

→ during final relaxation, -ve weight cycle condition is not met.

Answer  $\Rightarrow$  dist = 

0	-1	2	1
0	1	2	3

Tc  $\rightarrow O(V * E)$

Sc  $\rightarrow O(V)$



⑭ Negative weight cycle → Bellman Ford Algorithm.

→ To check the presence of negative weight cycle using Bellman Ford Algorithm.

TC →  $O(V * E)$

SC →  $O(V)$

code →

```
1 class Solution {
2 public:
3     int isNegativeWeightCycle(int n, vector<vector<int>>>edges){
4         vector<int>dis(n,INT_MAX);
5         // initially, dist to src is 0
6         dis[0] = 0;
7         // relax n-1 times
8         for(int i=0;i<n-1;i++)
9         {
10             for(auto edge:edges)
11             {
12                 int src = edge[0];
13                 int dest = edge[1];
14                 int wt = edge[2];
15                 if(dis[src]!=INT_MAX) // to avoid integer overflow
16                     dis[dest] = min(dis[dest],dis[src]+wt);
17             }
18         }
19         // final relaxation
20         for(auto edge:edges)
21         {
22             int src = edge[0];
23             int dest = edge[1];
24             int wt = edge [2];
25             if(dis[src]!=INT_MAX && dis[dest]>dis[src]+wt)
26                 return 1;
27         }
28         return 0;
29     }
30 };
```

## ⑮ Floyd Warshall Algorithm

→ All source shortest path & -ve edges allowed.

→ Since it's all source shortest path we need to run the loop for all nodes, considering it as intermediary vertex.

$$\rightarrow \text{cost}[i][j] = \min(\text{cost}[i][j], \text{cost}[i][k] + \text{cost}[k][j])$$

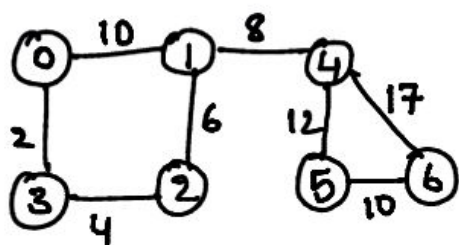
$$TC \rightarrow O(N^3) \quad SC \rightarrow O(N^2)$$

Code →

```
1 class Solution {
2     public:
3         void shortest_distance(vector<vector<int>>&matrix){
4             int V = matrix.size();
5             vector<vector<int>>costs(matrix.size(),vector<int>(matrix.size()));
6
7             for(int i=0;i<V;i++)
8                 for(int j=0;j<V;j++)
9                     costs[i][j] = matrix[i][j];
10
11             for(int k=0;k<V;k++)
12                 for(int i=0;i<V;i++)
13                     for(int j=0;j<V;j++){
14                         // if intermediate is not -1 then
15                         if(costs[i][k]!=-1 && costs[k][j]!=-1){
16                             if(costs[i][j]==-1)
17                                 costs[i][j] = costs[i][k]+costs[k][j];
18                             else
19                                 costs[i][j] = min(costs[i][j], costs[i][k]+costs[k][j]);
20                         }
21                     }
22
23             for(int i=0;i<V;i++)
24                 for(int j=0;j<V;j++)
25                     matrix[i][j] = costs[i][j];
26
27         }
28     };
```

# (16) Prim's Algorithm → Minimum Spanning Tree (MST)

Eg



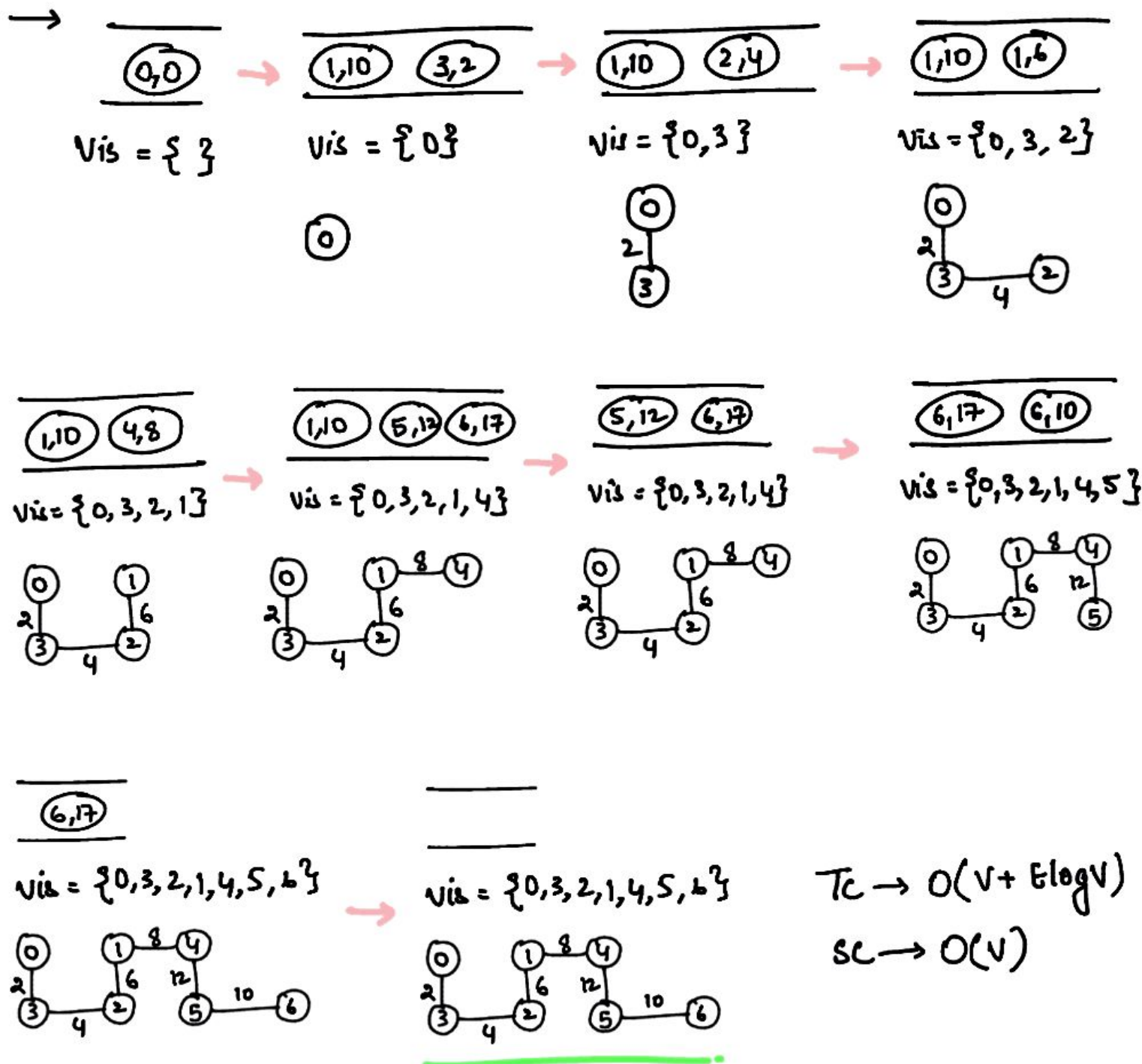
Vis = { }

PQ

node, weight

↑ returns node with lowest cost/weight.

\* To find MST, just push node along with its weight.



Tc →  $O(V + E \log V)$   
 Sc →  $O(V)$

Code →

```
1  class Solution
2  {
3      public:
4          //Function to find sum of weights of edges of the Minimum Spanning Tree.
5          int spanningTree(int V, vector<vector<int>> adj[])
6          {
7              int minCost = 0;
8              vector<int> costs(V, INT_MAX);
9              costs[0] = 0;
10             vector<bool> vis(V, false);
11             priority_queue<pair<int, int>, vector<pair<int, int>>, greater<pair<int, int>>> pq;
12             pq.push({0, 0}); // {cost, Node}
13
14             while(!pq.empty())
15             {
16                 pair<int, int> p = pq.top();
17                 int currNode = p.second;
18                 int currCost = p.first;
19                 pq.pop();
20
21                 if(vis[currNode]) continue;
22
23                 minCost += currCost;
24
25                 vis[currNode] = true;
26                 costs[currNode] = currCost;
27
28                 for(int i=0; i<adj[currNode].size(); i++)
29                 {
30                     int neighbourNode = adj[currNode][i][0];
31                     int neighbourNodeCost = adj[currNode][i][1];
32                     if(vis[neighbourNode]) continue;
33                     pq.push({neighbourNodeCost, neighbourNode});
34                 }
35             }
36             return minCost;
37         }
38     };
39
```



## 17) Min cost to connect all points

→ create graph with each node containing Wt & Node value

$$Wt = \text{abs}(X_i - X_j) + \text{abs}(Y_i - Y_j)$$

→ Perform Prim's algo.

$$T_c \rightarrow O(V + E \log V)$$

$$S_c \rightarrow O(V)$$

code →

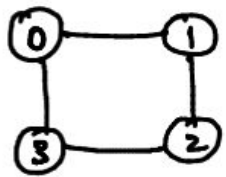
```
1
2 class Solution {
3 public:
4     int minCostConnectPoints(vector<vector<int>>& points) {
5
6         int n = points.size();
7         vector<vector<pair<int, int>>> graph(n);
8
9         for (int i = 0; i < n; i++) {
10             for (int j = 0; j < n; j++) {
11                 if (i == j) continue;
12                 graph[i].push_back({abs(points[i][0] - points[j][0]) + abs(points[i][1] - points[j][1]), j});
13             }
14         }
15
16         priority_queue<pair<int, int>, vector<pair<int, int>>, greater<pair<int, int>>> pq;
17         vector<bool> vis(n, false);
18         pq.push({0, 0}); // {cost, Node}
19
20         int ans = 0;
21         while (!pq.empty())
22         {
23             pair<int, int> p = pq.top();
24             int currNode = p.second;
25             int currCost = p.first;
26             pq.pop();
27
28             if (vis[currNode]) continue;
29             ans += currCost;
30             vis[currNode] = true;
31
32             for (int i = 0; i < graph[currNode].size(); i++)
33             {
34                 int neighbourNode = graph[currNode][i].second;
35                 int neighbourNodeCost = graph[currNode][i].first;
36                 if (vis[neighbourNode]) continue;
37                 pq.push({neighbourNodeCost, neighbourNode});
38             }
39         }
40
41         return ans;
42     }
43 };
44
```

### ⑮ Is Graph Bipartite

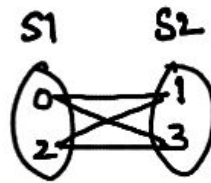
Bipartite graph is undirected graph, such that all vertices can be divided into 2 sets,  $S_1$  &  $S_2$  and no two vertices present in same set share an edge.

Eg

$n = 4$



then



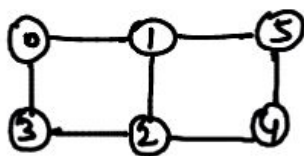
$\therefore$  the graph is bipartite.

$\Rightarrow$  for graph to be bipartite,

- it needs to be undirected acyclic graph (or)
- it needs to be even length cyclic graph

$\rightarrow$  we generally denote sets by coloring it, color = 0, 1.  
 $\downarrow \quad \downarrow$   
 $S_1 \quad S_2$

Eg  $n = 6$



$vis = \{ \}$      $S_1 = \{ \}$      $S_2 = \{ \}$

initially

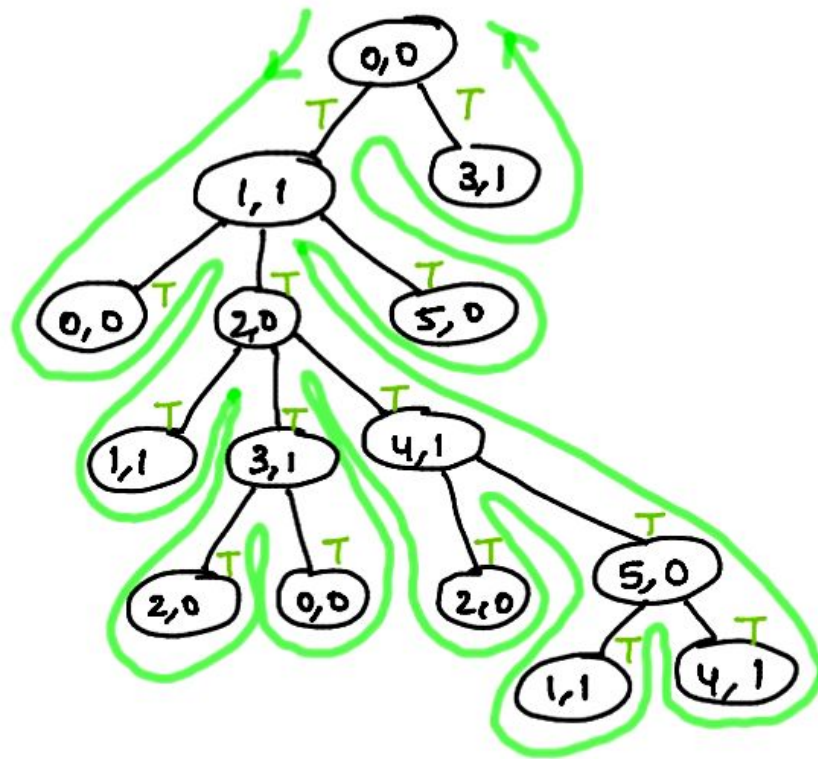


$\rightarrow$  at each vertex, check if it's visited or not.

$\rightarrow$  if visited then check if it's present in the intended set or not.

$\rightarrow$  if yes then return true, else false

$\rightarrow$  return AND of all the boolean values.



Code →

```
1 class Solution {
2 public:
3
4     bool isBipartite(vector<vector<int>>& graph) {
5
6         int n= graph.size();
7         vector<int> colors(n,-1);
8
9         for(int curr=0; curr<n ; curr++){
10             // if already colored then skip
11             if(colors[curr]!=-1) continue;
12             // check for even length cycle
13             if(hasEvenLengthCycle(graph, curr, 0, colors)==false) return false;
14         }
15         return true;
16     }
17
18     bool hasEvenLengthCycle(vector<vector<int>>& graph,int curr,int color,vector<int>&colors)
19     {
20         if(colors[curr]!=-1)
21             return colors[curr]==color;
22
23         // if not colored then color it
24         colors[curr] = color;
25
26         // check for neighbours
27         for(int neigh: graph[curr])
28         {
29             if(hasEvenLengthCycle(graph, neigh, 1-color, colors)==false)
30                 // 1- color will handle both changing colors 0 to 1 and 1 to 0
31                 return false;
32         }
33         return true;
34     }
35
36 };
```

## ①9 Possible Bipartition →

- create a graph using dislikes array.
- use previous problem's approach to solve it.

code →

$Tc \rightarrow O(V+E)$      $Sc \rightarrow O(V+E)$

```
1  class Solution {
2  public:
3
4      bool dfs(vector<int> graph[], int curr, vector<int>& color){
5
6          // if not colored then color
7          if(color[curr] == -1)
8              color[curr] = 1;
9
10         // process the neighbours and check their colors
11         for(auto neigh : graph[curr])
12         {
13             if(color[neigh] == -1)
14             {
15                 color[neigh] = 1 - color[curr];
16                 if(dfs(graph, neigh, color)==false) return false;
17             }
18             else if(color[neigh] == color[curr]) return false;
19         }
20         return true;
21     }
22
23     bool possibleBipartition(int n, vector<vector<int>>& dislikes) {
24         vector<int> color(n+1, -1);
25         vector<int> graph[n+1];
26
27         // populating the graph
28         for(auto edge : dislikes){
29             graph[edge[0]].push_back(edge[1]);
30             graph[edge[1]].push_back(edge[0]);
31         }
32
33         for(int i=1; i<=n; i++){
34             if(color[i] == -1)
35                 if(!dfs(graph, i, color)) return false;
36         }
37
38         return true;
39     }
40 };
```



②⑦ Disjoint Set  $\rightarrow$  UNION & FIND./getParent  
 $\downarrow$  helps in UNION of component/vertices.  $\rightarrow$  helps in finding parent of component

Ex ①  $\Rightarrow$  UNION(0,1)  $\rightarrow$  ①—②

Ex  $n=7$  initially every component is parent of itself



parent =

0	1	2	3	4	5	6
0	1	2	3	4	5	6

now  $\text{getParent}(2) = 2$ ,  $\text{getParent}(3) = 3$ .

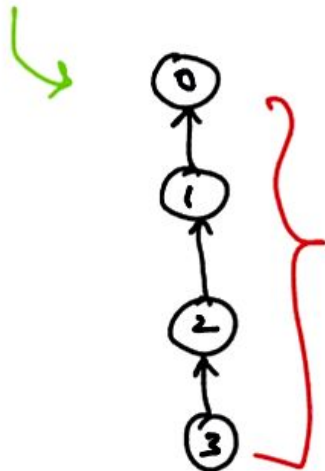
& if UNION(0,1)  $\Rightarrow$  ①—② &  $\text{parent}[1] = 0$

now  $\text{getParent}(1) = 0$

& UNION(1,2)  $\Rightarrow$  ①—②—③

UNION(2,3)  $\Rightarrow$  ①—②—③—④

&  $\text{getParent}(3) = 0$



This increases the recursive calls and the tree is unbalanced so we'll use rank array to store min. height tree for node.

$n = 7$  initially every component is parent of itself



parent =

0	1	2	3	4	5	6
0	1	2	3	4	5	6

rank =

0	0	0	0	0	0	0
0	1	2	3	4	5	6

$\Rightarrow \text{UNION}(0,1) \Rightarrow$  then find(0) & find(1) &  $0 \neq 1 \therefore$  diff components.  
as they are diff components find rank &  $\text{rank}[0] = \text{rank}[1] = 0$

$\therefore$  select either 0 or 1 & make it as root & inc the rank by 1



parent =

0	0	2	3	4	5	6
0	1	2	3	4	5	6

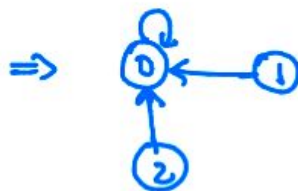
rank =

1	0	0	0	0	0	0
0	1	2	3	4	5	6

$\Rightarrow \text{UNION}(1,2) \Rightarrow \text{parent}(1) = 0$  &  $\text{parent}(2) = 2$   
now  $\text{rank}[0] = 1$  &  $\text{rank}[2] = 0$

as  $\text{rank}[0] > \text{rank}[2]$ ,

vertex 0 should be the parents  
& donot update rank if they are unequal.



parent =

0	0	0	3	4	5	6
0	1	2	3	4	5	6

rank =

1	0	0	0	0	0	0
0	1	2	3	4	5	6

Code →

```
1  class DisjSet {
2      int *rank, *parent, n;
3
4      public:
5      DisjSet(int n)
6      {
7          rank = new int[n];
8          parent = new int[n];
9          this->n = n;
10         makeSet();
11     }
12
13     void makeSet()
14     {
15         for (int i = 0; i < n; i++) {
16             parent[i] = i;
17         }
18     }
19
20     int find(int x)
21     {
22         // if x is not parent of itself then
23         // find parent recursively
24         if (parent[x] != x) {
25             parent[x] = find(parent[x]);
26         }
27         return parent[x];
28     }
29
30     void Union(int x, int y)
31     {
32         int xset = find(x);
33         int yset = find(y);
34
35         // if set of x and y are same then return
36         if (xset == yset) return;
37
38         // place the elements in small rank
39         if (rank[xset] < rank[yset]) {
40             parent[xset] = yset;
41         }
42         else if (rank[xset] > rank[yset]) {
43             parent[yset] = xset;
44         }
45         // if same rank then increment it
46         else {
47             parent[yset] = xset;
48             rank[xset] = rank[xset] + 1;
49         }
50     }
51 };
52
```

## ②1 Kruskal's Algorithm →

- This is used to find minimum spanning tree.
- can be implemented using Disjoint set.
- sort all the edges in ↑ order of weight.
- pick smallest edge & check if it contributes to cycle in graph
- if yes then discard else include.

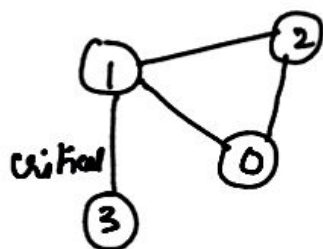
code →

```
1 class Graph {
2     vector<vector<int> > edgelist;
3     int V;
4
5 public:
6     Graph(int V) { this->V = V; }
7
8     void addEdge(int x, int y, int w)
9     {
10         edgelist.push_back({ w, x, y });
11     }
12
13     void kruskals_mst()
14     {
15         // 1. Sort all edges
16         sort(edgelist.begin(), edgelist.end());
17
18         // Initialize the DSU - DisjointSet
19         DSU s(V);
20         int ans = 0;
21         for (auto edge : edgelist) {
22             int w = edge[0];
23             int x = edge[1];
24             int y = edge[2];
25             // take that edge in MST if it does form a cycle
26             if (s.find(x) != s.find(y)) {
27                 s.union(x, y);
28                 ans += w;
29                 cout << x << " -- " << y << " == " << w
30                     << endl;
31             }
32         }
33         cout << "Minimum Cost Spanning Tree: " << ans;
34     }
35 };
```



## ②② Critical Connection in a Network →

Eg  $n=4$  edges =  $[[0,1], [1,2], [2,0], [1,3]]$



→ Critical connection is a connection, when removed from graph, would result in breaking graph into different components.

Here if  $[1,3]$  is removed then graph becomes disconnected.

### Approach 1

- Remove one edge each time
- Perform dfs
- If all vertices are not visited then
- Removed edge is a critical connection.

### Approach 2

- initialise discovery time for vertex distime array & min time for vertex to be discovered. minime array with -1.
- perform dfs from one node
- if neighbour == parent then continue
- else if neighbour is already visited then  
$$\text{minime}[\text{curr}] = \min(\text{minime}[\text{curr}], \text{distime}[\text{neigh}])$$
- while returning  $\text{minime}[\text{curr}] = \min(\text{minime}[\text{curr}], \text{minime}[\text{neigh}])$   
& at any point if  $\text{distime}[\text{curr}] < \text{minime}[\text{neigh}]$   
**This indicates critical connection**

code →

```
1  class Solution {
2  public:
3
4      vector<vector<int>> criticalConnections(int n, vector<vector<int>>& connections) {
5          vector<int> graph[n];
6          for(vector<int> edge: connections){
7              int u = edge[0];
8              int v = edge[1];
9              graph[u].push_back(v);
10             graph[v].push_back(u);
11         }
12         return findCriticalConnections(n, graph);
13     }
14
15     vector<vector<int>> findCriticalConnections(int n, vector<int> graph[]){
16         vector<int> disTime(n,-1);
17         vector<int> lowTime(n,-1);
18         int time = 0;
19         vector<vector<int>> answer;
20         tarjansDFS(graph, 0, -1, disTime, lowTime, time, answer);
21         return answer;
22     }
23
24     void tarjansDFS(vector<int> graph[], int curr, int parent, vector<int>&disTime,
25     vector<int> &lowTime, int &time, vector<vector<int>> &answer){
26
27         disTime[curr] = time;
28         lowTime[curr] = time;
29         time += 1;
30
31         for(int neigh: graph[curr]){
32             if(neigh == parent) continue;
33
34             if(disTime[neigh] != -1){
35                 lowTime[curr] = min(lowTime[curr], disTime[neigh]);
36                 continue;
37             }
38
39             tarjansDFS(graph, neigh, curr, disTime, lowTime, time, answer);
40             lowTime[curr] = min(lowTime[curr], lowTime[neigh]);
41
42             if(disTime[curr] < lowTime[neigh]){
43                 vector<int> temp;
44                 temp.push_back(curr);
45                 temp.push_back(neigh);
46                 answer.push_back(temp);
47             }
48         }
49         return;
50     }
51
52     };
```

# Dynamic Programming - 1

- Karun Karthik

## Contents

0. Introduction
1. Climbing Stairs
2. Fibonacci Number
3. Min Cost Climbing Stairs
4. House Robber
5. House Robber - II
6. Nth Tribonacci Number
7. 0-1 Knapsack
8. Partition Equal Subset Sum
9. Target Sum
10. Count no of Subsets with given Difference
11. Delete and Earn
12. Knapsack with Duplicate Items
13. Coin Change - II
14. Coin Change
15. Rod cutting

# Introduction

Dynamic programming is a technique to solve problems by breaking it down into a collection of sub-problems, solving each of those sub-problems just once and storing these solutions inside the cache memory in case the same problem occurs the next time.

Dynamic Programming is mainly an optimization over plain recursion .

Wherever we see a recursive solution that has repeated calls for same inputs, we can optimize it using Dynamic Programming.

This simple optimization reduces the time complexities from exponential to polynomial.

There are two different ways to store our values so that they can be reused at a later instance. They are as follows:

1. Memoization or the Top Down Approach.
2. Tabulation or the Bottom Up approach.

In Memoization we start from the extreme state and compute result by using values that can reach the destination state i.e the base state.

In Tabulation we start from the base state and then compute results all the way till the extreme state.

Note: To store the intermediate results we can use Array, Matrix, Hashmap etc., all we need is data storage and retrieval with a specific key.

How to find the use case of Dynamic Programming?

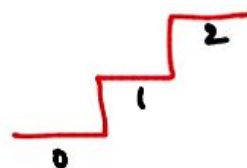
You can use DP if the problem can be,

1. Divided into sub-problems
2. Solved using a recursive solution
3. Containing repetitive sub-problems

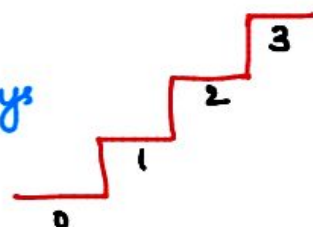


① Climbing Stairs → Given a value 'N', find the number of ways to reach N & jumps possible are ONE or TWO.

eg  $n=2 \Rightarrow$

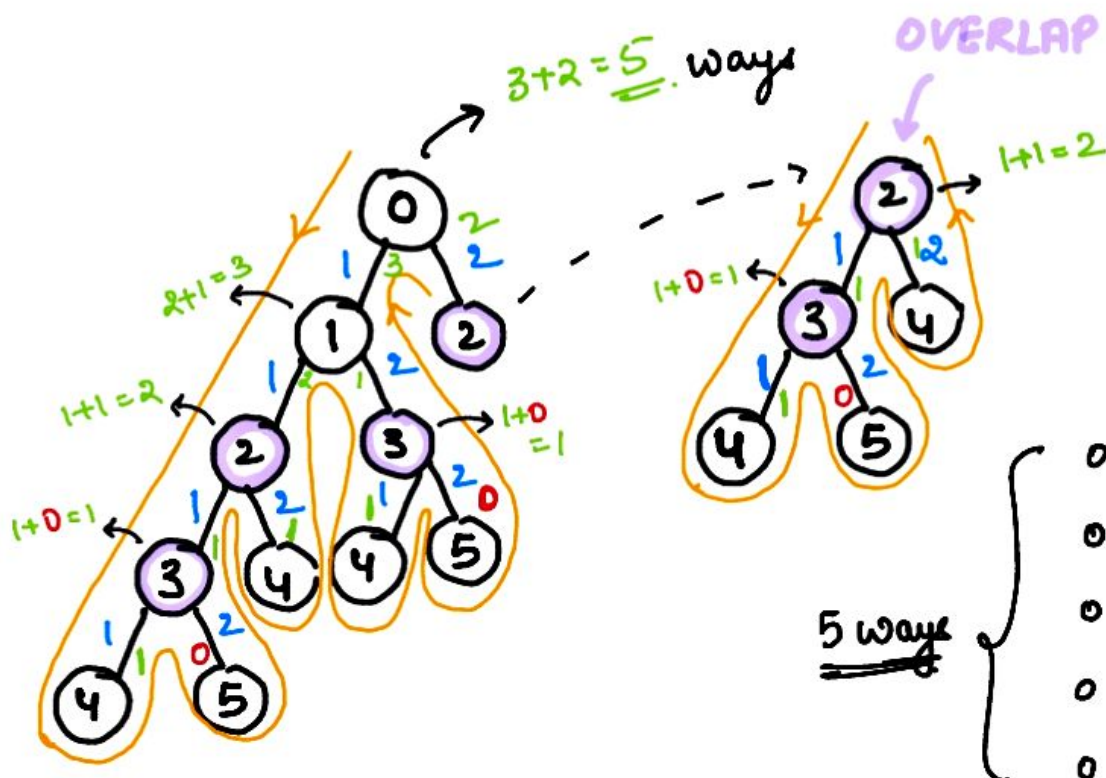
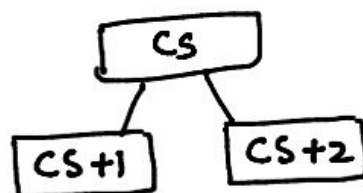
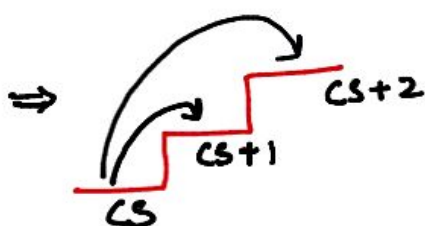
$$\begin{array}{l} 0 \xrightarrow{1} 1 \xrightarrow{1} 2 \\ 0 \xrightarrow{2} 2 \end{array} \left. \vphantom{\begin{array}{l} 0 \xrightarrow{1} 1 \xrightarrow{1} 2 \\ 0 \xrightarrow{2} 2 \end{array}} \right\} \begin{array}{l} \text{for } N=2 \\ \text{we have 2 ways} \end{array}$$


$n=3 \Rightarrow$

$$\begin{array}{l} 0 \xrightarrow{1} 1 \xrightarrow{1} 2 \xrightarrow{1} 3 \\ 0 \xrightarrow{1} 1 \xrightarrow{2} 3 \\ 0 \xrightarrow{2} 2 \xrightarrow{1} 3 \end{array} \left. \vphantom{\begin{array}{l} 0 \xrightarrow{1} 1 \xrightarrow{1} 2 \xrightarrow{1} 3 \\ 0 \xrightarrow{1} 1 \xrightarrow{2} 3 \\ 0 \xrightarrow{2} 2 \xrightarrow{1} 3 \end{array}} \right\} \begin{array}{l} \text{for } N=3 \\ \text{we have 3 ways} \end{array}$$


$n=4$

→ for every stair we have 2 cases i.e.



if  $CS == n$   
return 1  
if  $CS > n$   
return 0

5 ways

$0 \xrightarrow{1} 1 \xrightarrow{1} 2 \xrightarrow{1} 3 \xrightarrow{1} 4$   
 $0 \xrightarrow{1} 1 \xrightarrow{1} 2 \xrightarrow{2} 4$   
 $0 \xrightarrow{1} 1 \xrightarrow{2} 3 \xrightarrow{1} 4$   
 $0 \xrightarrow{2} 2 \xrightarrow{1} 3 \xrightarrow{1} 4$   
 $0 \xrightarrow{2} 2 \xrightarrow{2} 4$

\* Here we can see for (2), (3)

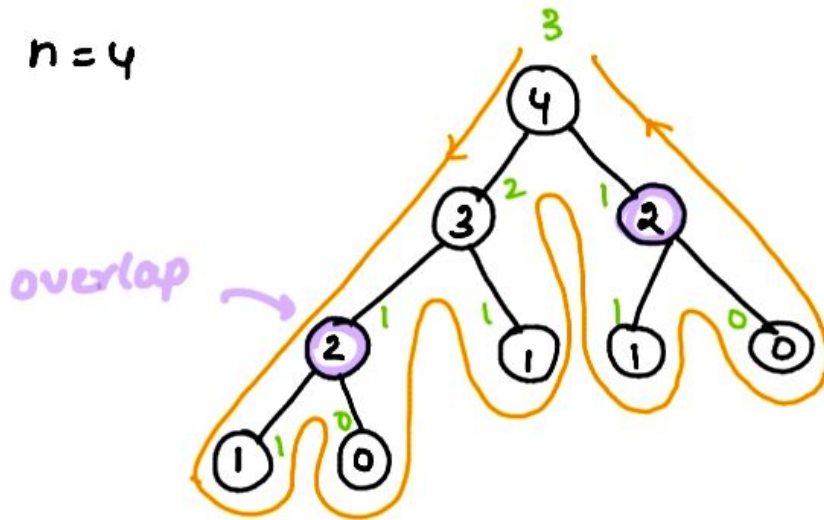
the subproblem is being done multiple times, we can solve using dp.

code →

```
1  class Solution {
2  public:
3      int totalWays(int currentStair, int targetStair, unordered_map<int,int> &memo){
4
5          if(currentStair==targetStair){
6              return 1;
7          }
8
9          if(currentStair > targetStair){
10             return 0;
11         }
12
13         int currentKey = currentStair;
14
15         if(memo.find(currentKey)!=memo.end()){
16             return memo[currentKey];
17         }
18
19         int oneStep = totalWays(currentStair+1, targetStair, memo);
20         int twoStep = totalWays(currentStair+2, targetStair, memo);
21
22         memo[currentKey] = oneStep+twoStep;
23
24         return oneStep+twoStep;
25     }
26
27     int climbStairs(int n) {
28         unordered_map<int,int> memo;
29         return totalWays(0,n,memo);
30     }
31 }
32 };
```

② Fibonacci Number  $\rightarrow f(n) = f(n-1) + f(n-2)$  &  $f(0) = 0$   
 $f(1) = 1$   
 $n \geq 0$

Ex  $\rightarrow n = 4$



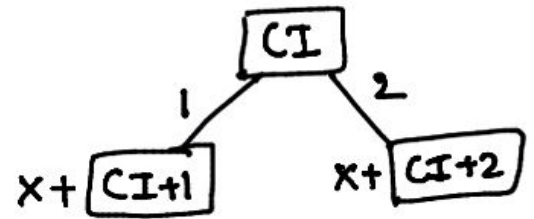
code  $\rightarrow$

```
1 class Solution {
2 public:
3     int helper(int n, unordered_map<int,int>&memo){
4
5         if(n<=1){
6             return n;
7         }
8
9         int currentKey = n;
10
11         if(memo.find(currentKey)!=memo.end()){
12             return memo[currentKey];
13         }
14
15
16         int a = helper(n-1,memo);
17         int b = helper(n-2,memo);
18
19         memo[currentKey] = a+b;
20         return memo[currentKey];
21     }
22
23
24     int fib(int n) {
25         unordered_map<int,int>memo;
26         return helper(n,memo);
27     }
28 };
```

### ③ Min Cost Climbing Stairs →

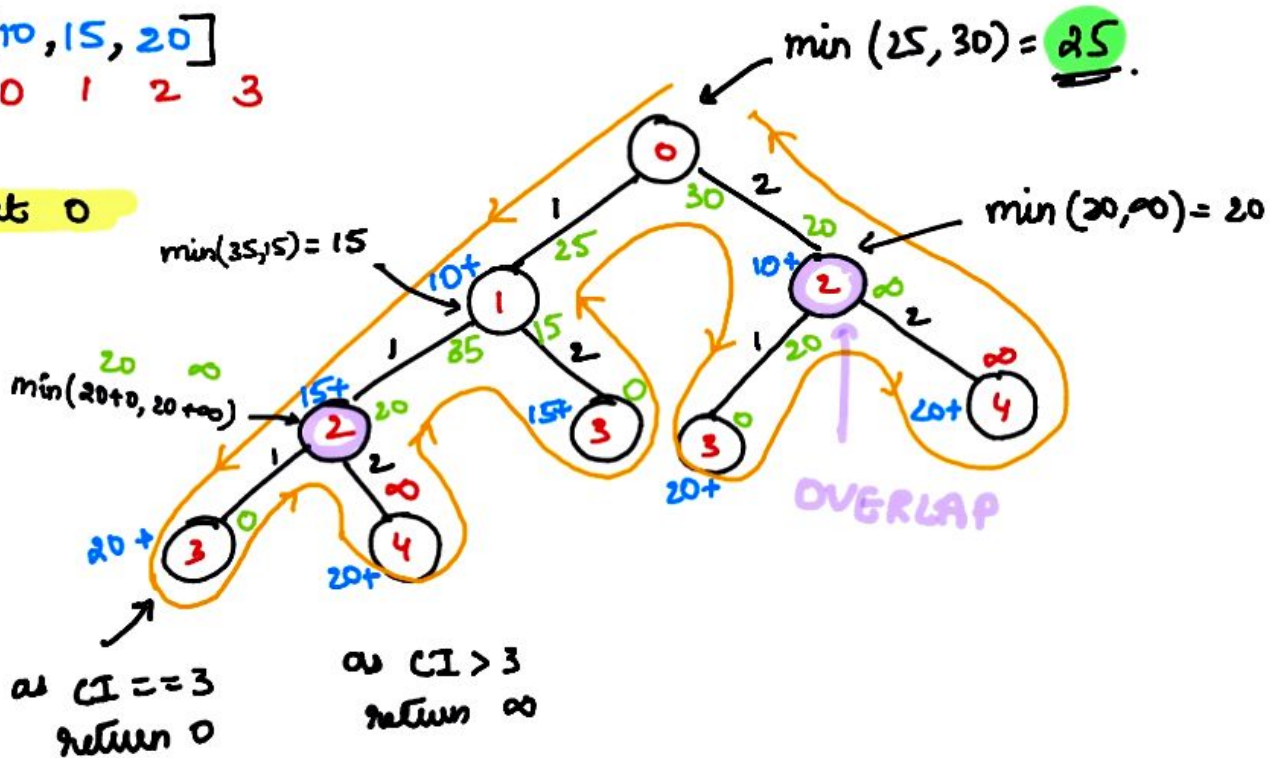
Given costs array, find min cost to reach the end, starting from 0 or 1 & making 1 or 2 jumps.

$$\therefore \text{costs} = [ \_ \_ \underset{\substack{\uparrow \\ \text{CI}}}{x} \_ \_ ]$$

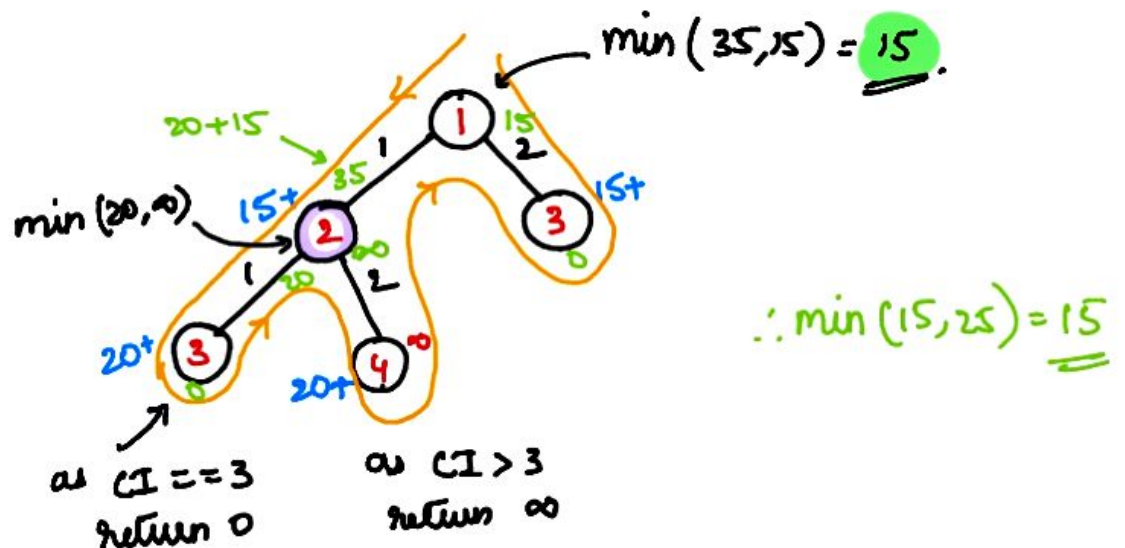


Eg cost = [10, 15, 20]  
0 1 2 3

starting at 0



starting at 1





Code →

```
1  class Solution {
2  public:
3
4      int minCost(vector<int>&cost, int currentIndex, unordered_map<int,int> &m){
5
6          if(currentIndex == cost.size()){
7              return 0;
8          }
9
10         if(currentIndex > cost.size()){
11             return 1000;    // large values, serves as INFINITY
12         }
13
14         if(m.find(currentIndex)!=m.end()){
15             return m[currentIndex];
16         }
17
18         int oneJump = cost[currentIndex] + minCost(cost,currentIndex+1, m);
19         int twoJump = cost[currentIndex] + minCost(cost,currentIndex+2, m);
20
21         m[currentIndex] = min(oneJump, twoJump);
22         return m[currentIndex];
23     }
24
25     int minCostClimbingStairs(vector<int>& cost) {
26         unordered_map<int,int> m;
27         return min( minCost(cost,0,m), minCost(cost,1,m));
28     }
29 };
```

④ House Robber → Given an array of no. representing money, find max amount, that can be robbed without choosing the adjacent houses.

Eg  $\text{nums} = [2, 7, 9, 3, 1]$   $\Rightarrow \text{max amount} = 2 + 9 + 1 = 12$ .  
 $0 \rightarrow 2 \rightarrow 4$

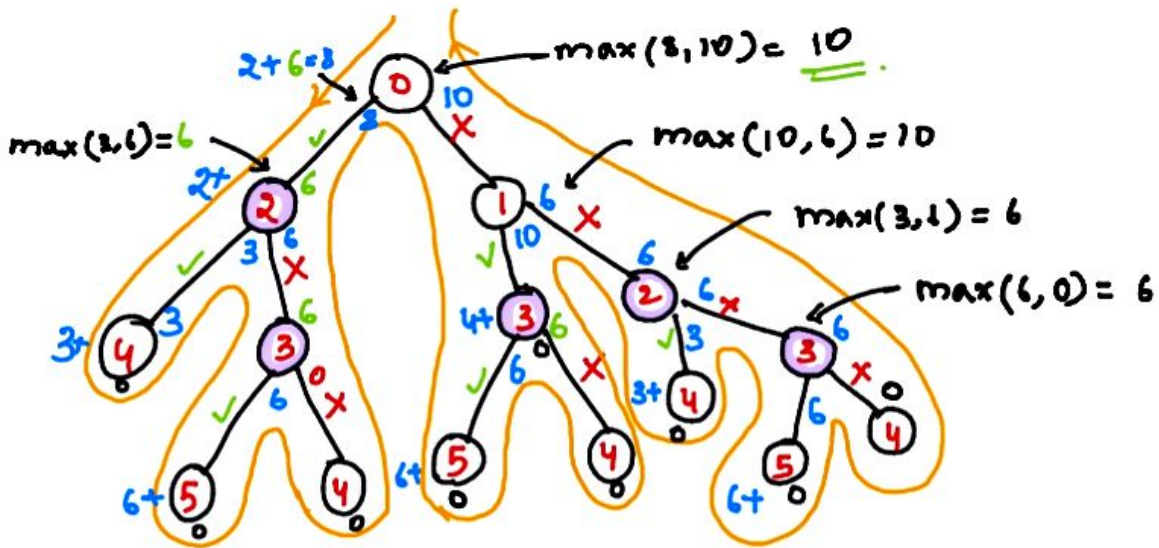
→ If robber robs house, then amount = nums[CI] & CI = CI + 2 <sup>to avoid adjacent</sup>  
 else CI = CI + 1

ii  $nums = [ \text{---} \text{---} \text{---} \text{---} ]$

as he robs  
he gets money  
↓  
 $num[C] +$

```
graph TD
    C[C] -- "rob ✓" --> C2[C+2]
    C[C] -- "no rob ✗" --> C1[C+1]
```

5.  $[2, 4, 3, 6]$   
0 1 2 3



```

    if (CI > 3)
        return 0
    }
}

```

∴ at every node find  $\max(\text{left}, \text{right})$   
 & add it's value to the  $\text{nums}[i]$   
 if selected, else continue

Code →

```
1  class Solution {
2  public:
3
4      int helper(vector<int>&nums, int currentIndex, unordered_map<int,int>&m){
5
6          if(currentIndex >= nums.size()){
7              return 0;
8          }
9
10         int currentKey = currentIndex;
11
12         if(m.find(currentKey)!=m.end()){
13             return m[currentKey];
14         }
15
16         int rob = nums[currentKey] + helper(nums, currentIndex+2, m);
17         int noRob = helper(nums, currentIndex+1, m);
18
19         m[currentIndex] = max(rob, noRob);
20
21         return m[currentIndex];
22     }
23
24     int rob(vector<int>& nums) {
25         unordered_map<int,int> m;
26         return helper(nums,0,m);
27     }
28 };
```

## ⑤ House Robber - II →

In this problem, the approach will be similar to previous one, but the houses are in circle, which means that

- \* if we start from 1<sup>st</sup> house, then we can't rob the last house.

- \* if we start from 2<sup>nd</sup> house, then we can rob the last house.

- \* and return max value between 1<sup>st</sup> house & 2<sup>nd</sup> house

- \* if only 1 house is present, then rob it directly.

Code →

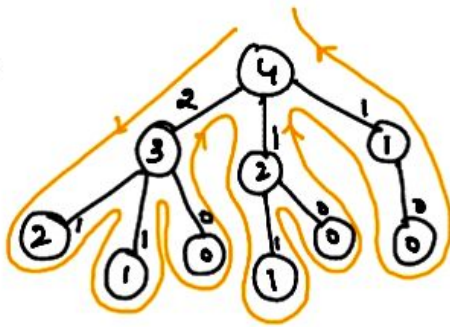
```
1 class Solution {
2 public:
3
4     int helper(vector<int>&nums, int currentIndex, int lastIndex, unordered_map<int,int>&m){
5
6         if(currentIndex > lastIndex){
7             return 0;
8         }
9
10        int currentKey = currentIndex;
11
12        if(m.find(currentKey)!=m.end()){
13            return m[currentKey];
14        }
15
16        int rob = nums[currentKey] + helper(nums, currentIndex+2, lastIndex, m);
17        int noRob = helper(nums, currentIndex+1, lastIndex, m);
18
19        m[currentIndex] = max(rob, noRob);
20
21        return m[currentIndex];
22    }
23
24    int rob(vector<int>& nums) {
25
26        int n = nums.size();
27        if(n==1) return nums[0];
28
29        unordered_map<int,int> memo1,memo2;
30        // we can start robbing from 2 houses
31        int firstHouse = helper(nums, 0, n-2, memo1);
32        int secondHouse = helper(nums, 1, n-1, memo2);
33        return max(firstHouse, secondHouse);
34    }
35 }
36 };
```



⑥ N-th Tribonacci → given  $n$ , find  $T_n$

$$T_{n+3} = T_n + T_{n+1} + T_{n+2} \quad \text{if } n \geq 0 \quad T_0 = 0, T_1 = 1, T_2 = 1.$$

Eg  $n = 4$



4 (2+1+1)

code →

```
1 class Solution {
2 public:
3
4     int helper(int n, unordered_map<int,int> &m){
5         if(n<=1){
6             return n;
7         }
8
9         if(n==2){
10             return 1;
11         }
12
13         int currentNum = n;
14
15         if(m.find(currentNum)!=m.end()){
16             return m[currentNum];
17         }
18
19         int a = helper(n-1,m);
20         int b = helper(n-2,m);
21         int c = helper(n-3,m);
22
23         m[currentNum] = a+b+c;
24
25         return m[currentNum];
26     }
27
28     int tribonacci(int n) {
29         unordered_map<int,int>m;
30         return helper(n,m);
31     }
32 };
```

⑦ 0-1 Knapsack Problem → find max profit such that the weight of all items  $\leq$  capacity.

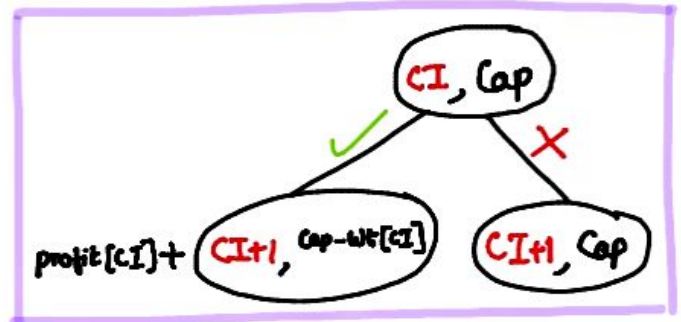
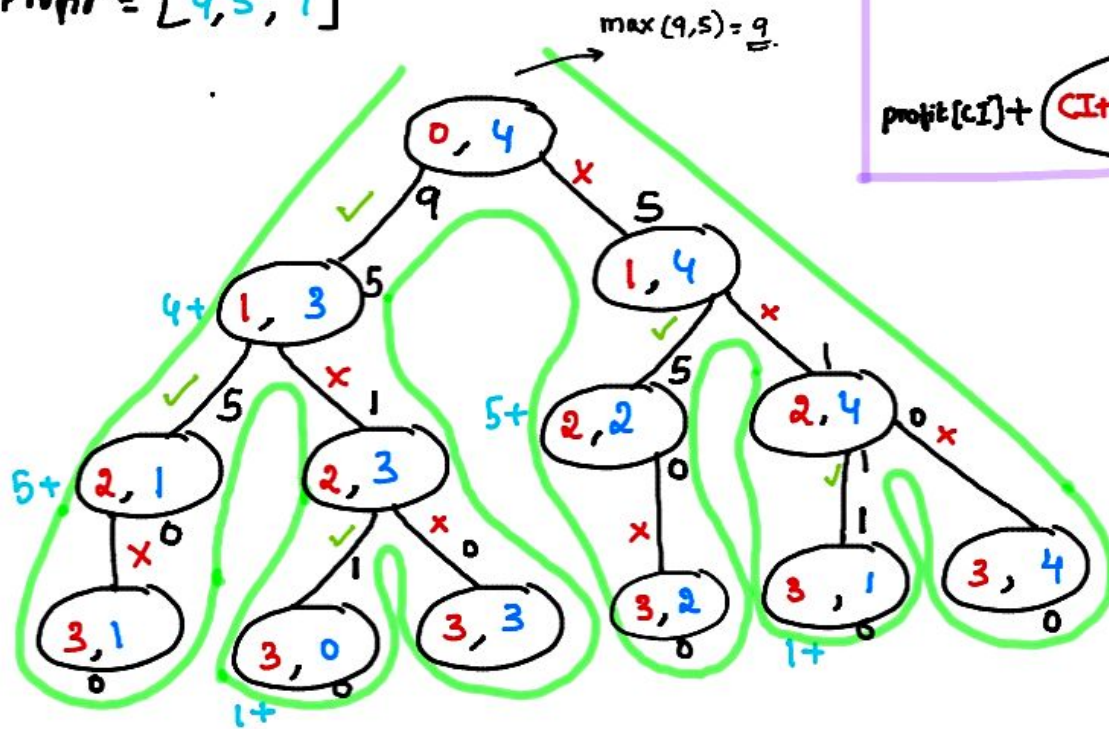
Wt = <sup>0 1 2 3</sup> [3, 4, 6, 5]  
 profits = [2, 3, 1, 4] → if we select 0 & 3,  
 then total weight =  $wt[0] + wt[3]$   
 $= 3 + 5 = 8$ .

capacity = 8

& profits are  $2 + 4 = \underline{6}$  That's max profit possible

Eg

Wt = <sup>0 1 2</sup> [1, 2, 3]      capacity = 4  
 Profit = [4, 5, 1]



∴ at every step,

- if selecting an index then reduce capacity by  $wt[CI]$  & add profit  $[CI]$  to result
- if not selecting, increment  $CI$  by 1
- find  $\max(\text{left}, \text{right})$

code →

```
1  class Solution
2  {
3      public:
4
5      int helper(int W, int wt[], int val[], int n, int curr,
6                  unordered_map<string,int> &memo){
7          if(curr==n) return 0;
8
9          // Instead of Matrix we can use strings as unique keys
10         string currKey = to_string(curr)+"_"+to_string(W);
11
12         if(memo.find(currKey)!=memo.end()) return memo[currKey];
13
14         int currWt = wt[curr];
15         int currVal = val[curr];
16
17         int selected = 0;
18         if(currWt<=W){
19             selected = currVal + helper(W-currWt, wt, val, n, curr+1, memo);
20         }
21
22         int notSelected = helper(W, wt, val, n, curr+1, memo);
23
24         memo[currKey] = max(selected, notSelected);
25         return memo[currKey];
26     }
27
28
29     int knapSack(int W, int wt[], int val[], int n)
30     {
31         unordered_map<string,int> memo;
32         return helper(W, wt, val, n, 0, memo);
33     }
34 };
```

⑧ Partition Equal Subset Sum →

Given an array, find if it can be divided into two subsets whose sum is equal.

Eg  $\text{nums} = [1, 5, 11, 5]$  can be divided into  $S1 = \{1, 5, 5\}$  &  $S2 = \{11\}$   
&  $\text{sum of } S1 = \text{sum of } S2 \therefore \text{returns True.}$

$\therefore$  initially find sum of elements in array.

1) if sum is odd then return False

2) if sum is even, then proceed.

→ find a subset whose value ==  $\text{sum}/2$

which means that the other subset will have value  $= \text{sum}/2$ .

→ let say  $ts = \text{sum}/2$  ( $ts$  is target sum)

→ At every index, we have 2 choices

1) if we select then  $ts = ts - \underset{\substack{\uparrow \\ CI = CI + 1}}{num}[CI]$

2) if we do not select then  $ts = ts$  (ie remains same)  
 $CI = CI + 1$

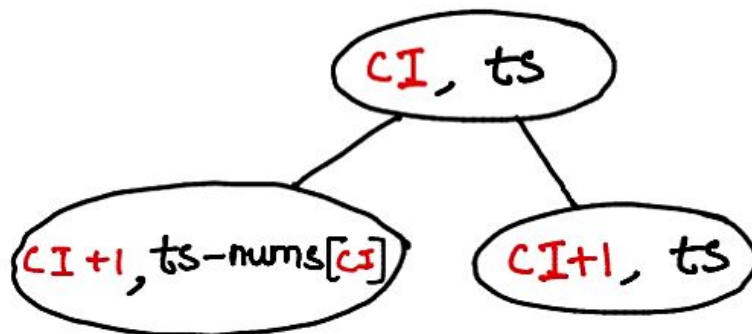
3) return OR of left & right branch.



$\Rightarrow \text{nums} = [1, 5, 11, 5]$

$\text{sum} = 22$

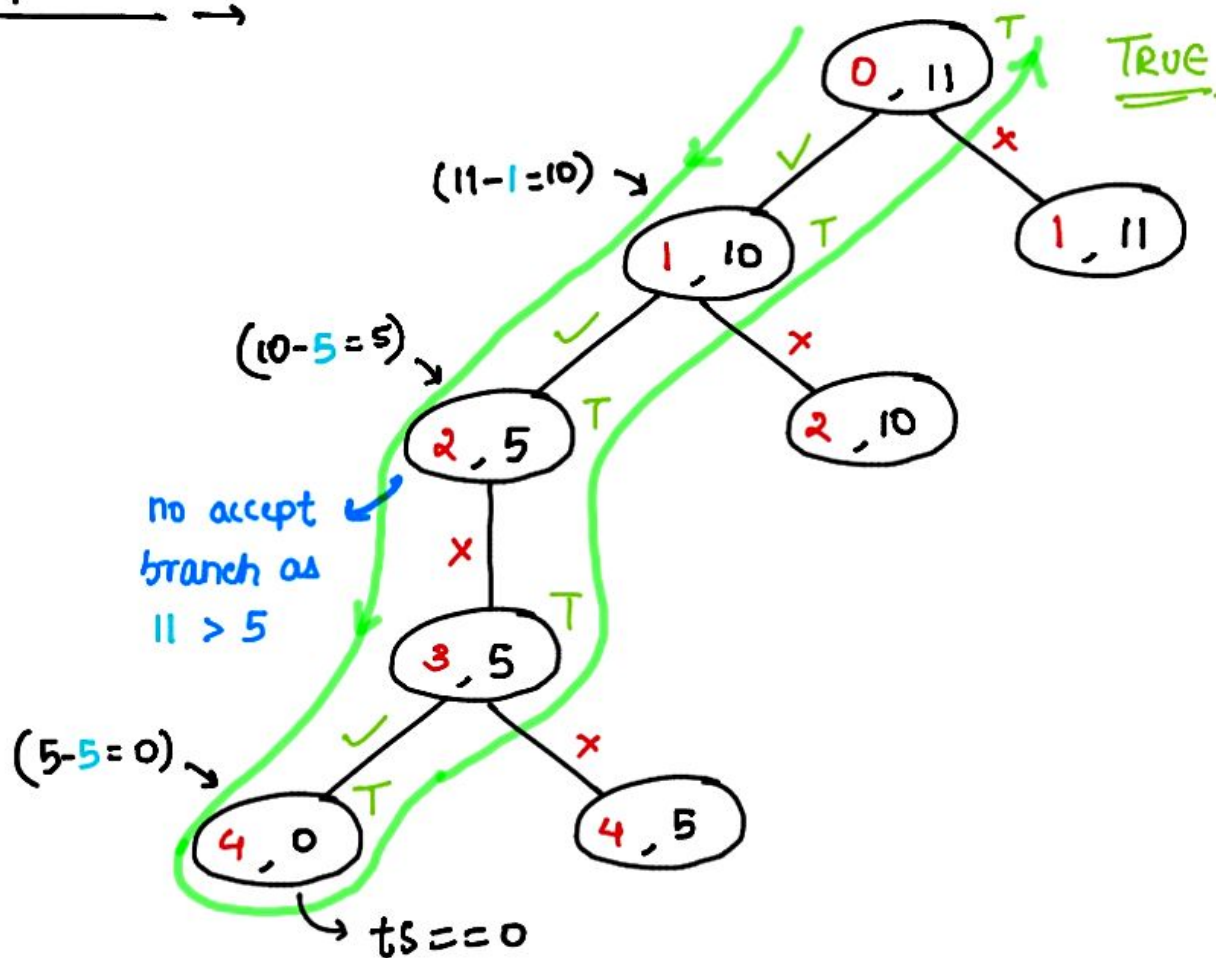
$\text{ts} = 11$



Here  $\text{sum} \% 2 == 0$

$\therefore$  dividing into 2 subsets is possible.

Explanation  $\rightarrow$



$\Rightarrow$  that subset is found  
 & return True

as we are using OR, one True branch is sufficient

Code →

```
1  class Solution {
2  public:
3
4      bool isPossible(int targetSum, int currentIndex, vector<int>&nums,
5                      unordered_map<string, bool> &memo){
6
7          if(targetSum == 0)
8              return true;
9
10         if(currentIndex >= nums.size())
11             return false;
12
13         string currentKey = to_string(currentIndex)+"_"+to_string(targetSum);
14
15         if(memo.find(currentKey)!=memo.end()){
16             return memo[currentKey];
17         }
18
19         bool possible = false;
20
21         if(nums[currentIndex]<=targetSum)
22             possible = isPossible(targetSum-nums[currentIndex], currentIndex+1, nums, memo);
23
24         // if already Possible then return True directly
25         if(possible){
26             memo[currentKey] = possible;
27             return true;
28         }
29
30         bool notPossible = isPossible(targetSum, currentIndex+1, nums, memo);
31
32         memo[currentKey] = possible||notPossible;
33         return memo[currentKey];
34     }
35
36     bool canPartition(vector<int>& nums) {
37
38         int total = 0;
39         for(auto it:nums) total+= it;
40
41         if(total%2!=0) return false;
42
43         unordered_map<string, bool> memo;
44         return isPossible(total/2,0, nums,memo);
45     }
46 };
```

## 9) Target Sum →

Given an array & target, find the number of ways to reach target by using + or - before each element in array.

Ex

Input: nums = [1,1,1,1,1], target = 3

Output: 5

Explanation: There are 5 ways to assign symbols to make the sum of nums be target 3.

$$-1 + 1 + 1 + 1 + 1 = 3$$

$$+1 - 1 + 1 + 1 + 1 = 3$$

$$+1 + 1 - 1 + 1 + 1 = 3$$

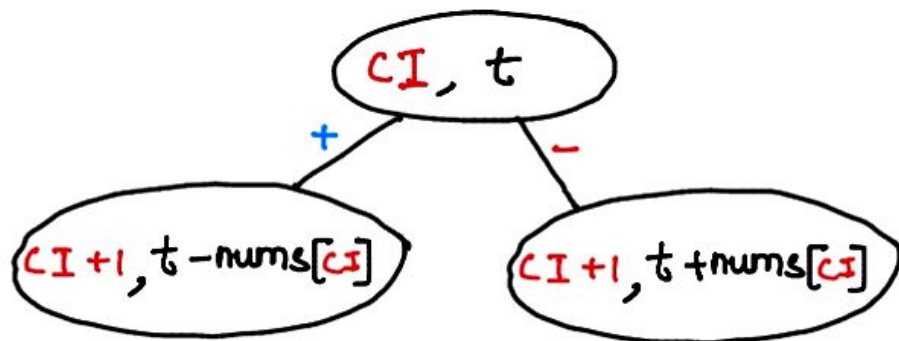
$$+1 + 1 + 1 - 1 + 1 = 3$$

$$+1 + 1 + 1 + 1 - 1 = 3$$

→ at every index we can use + or - sign

if + then  $t = t - (+ \text{nums}[CI]) \Rightarrow t - \text{nums}[CI]$

if - then  $t = t - (- \text{nums}[CI]) \Rightarrow t + \text{nums}[CI]$



→ at every node, return the sum of values from left & right. Because we need to find the total number of ways.

code →

```
1 class Solution {
2 public:
3     int totalWays(int currentIndex, vector<int>&nums, int target, unordered_map<string,int> &memo){
4
5         if(target==0 and currentIndex==nums.size()){
6             return 1;
7         }
8
9         if(currentIndex>=nums.size() and target!=0){
10             return 0;
11         }
12
13         string key = to_string(currentIndex)+"_"+to_string(target);
14
15         if(memo.find(key)!=memo.end()){
16             return memo[key];
17         }
18
19         int plus = totalWays(currentIndex+1, nums, target-nums[currentIndex],memo);
20
21         int minus = totalWays(currentIndex+1, nums, target+nums[currentIndex],memo);
22
23         memo[key] = plus+minus;
24
25         return plus+minus;
26     }
27
28     int findTargetSumWays(vector<int>& nums, int target) {
29         unordered_map<string,int> memo;
30         return totalWays(0,nums,target,memo);
31     }
32 };
```



⑩ Count number of subsets with given difference →

→ This is similar to Target Sum.

given the difference between two subsets, and an array  
find no. of subsets with the difference.

Approach →

Let's say  $s_1 - s_2 = \text{difference (given)}$  — ①

we can calculate sum of every element, say sum

& it can be said that for 2 subsets  $s_1$  &  $s_2$

$s_1 + s_2 = \text{sum}$ . — ②

Now ① + ②  $\Rightarrow 2(s_1) = \text{difference} + \text{sum}$

$$s_1 = (\text{difference} + \text{sum}) / 2.$$

→ Implement Target Sum with target value =  $s_1$ .

## ⑪ Delete and Earn →

You are given an integer array `nums`. You want to maximize the number of points you get by performing the following operation any number of times:

- Pick any `nums[i]` and delete it to earn `nums[i]` points. Afterwards, you must delete **every** element equal to `nums[i] - 1` and **every** element equal to `nums[i] + 1`.

Return the **maximum number of points** you can earn by applying the above operation some number of times.

Eg `nums = [2, 2, 3, 3, 3, 4]`

→ if we start deleting **2**, then  $\text{result} = 2 + 2 = 4$

then `nums = [3, 3, 3, 4]`

& we need to delete all  $2+1$  &  $2-1 \Rightarrow \text{nums} = [4]$

→ if we delete 4, then  $\text{result} = 4 + 4 = \underline{8}$ .

& `nums = []`

(or)

→ if we start deleting **3**, then  $\text{result} = 3 + 3 + 3 = 9$ .

then `nums = [2, 2, 4]`

& we need to delete all  $3-1$  &  $3+1 \Rightarrow \text{nums} = []$

$\therefore \text{result} = \underline{9}$ .

(or)

→ if we start deleting **4**, then  $\text{result} = 4$

then `nums = [2, 2, 3, 3, 3]`

& we need to delete all  $4+1$  &  $4-1 \Rightarrow \text{nums} = [2, 2]$

→ if we delete 2, then  $\text{result} = 4 + 4 = \underline{8}$ .

& `nums = []`

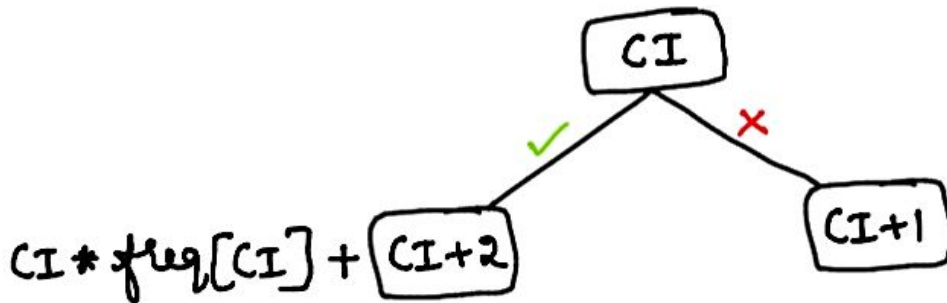
→ we can store frequency of each element and use the similar approach

nums = [2, 2, 3, 3, 3, 4]

freq = 

0	0	2	3	1
---	---	---	---	---

  
          0    1    2    3    4  
          ↑  
          CI



Code →

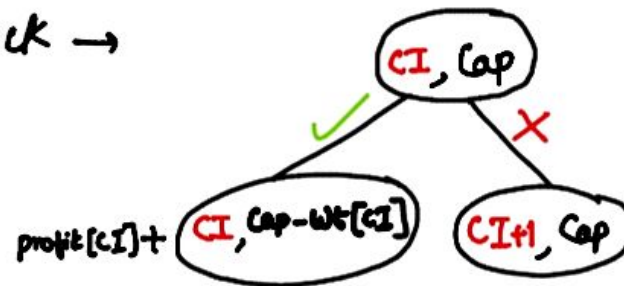
```
1 class Solution {
2 public:
3
4     int maxPoints(vector<int>& freq, int currentIndex, unordered_map<int,int>&memo){
5
6         if(currentIndex >= freq.size()) return 0;
7
8         int key = currentIndex;
9
10        if(memo.find(key) != memo.end()) return memo[key];
11
12        int Delete = currentIndex*freq[currentIndex] + maxPoints(freq, currentIndex+2, memo);
13        int NotDelete = maxPoints(freq, currentIndex+1, memo);
14
15        memo[key] = max(Delete, NotDelete);
16
17        return memo[key];
18    }
19
20    int deleteAndEarn(vector<int>& nums) {
21
22        int maxi = *max_element(nums.begin(), nums.end());
23        vector<int> freq(maxi+1, 0);
24
25        for(auto i: nums) freq[i]++;
26
27        unordered_map<int,int> memo;
28
29        return maxPoints(freq, 0, memo);
30    }
31 };
```

12 Unbounded Knapsack → Similar to 0-1 Knapsack but allows us to choose an item more than once

eg  
wt = [2, 1]  
values = [1, 1]  
capacity = 3

if bounded knapsack then profit = 2.  $(2, 1)$   
if unbounded knapsack then profit = 3.  $(1, 1, 1)$

for unbounded knapsack →



Code →

```
1 class Solution{
2 public:
3     int helper(int W, int wt[], int val[], int N, int curr, vector<vector<int>>&memo){
4
5         if(W==0) return 0;
6         if(curr==N) return 0;
7
8         if(memo[curr][W]!=-1) return memo[curr][W];
9
10        int currWt = wt[curr];
11        int currVal = val[curr];
12
13        int selected = 0;
14        if(currWt<=W){
15            selected = currVal + helper(W-currWt, wt, val, N, curr, memo);
16        }
17
18        int notSelected = helper(W, wt, val, N, curr+1, memo);
19
20        memo[curr][W] = max(selected, notSelected);
21        return memo[curr][W];
22    }
23
24    int knapSack(int N, int W, int val[], int wt[])
25    {
26        vector<vector<int>> memo( N , vector<int> (W+1, -1));
27        return helper(W, wt, val, N, 0, memo);
28    }
29 }
```



### ⑬ Coin Change II → (Similar to unbounded knapsack)

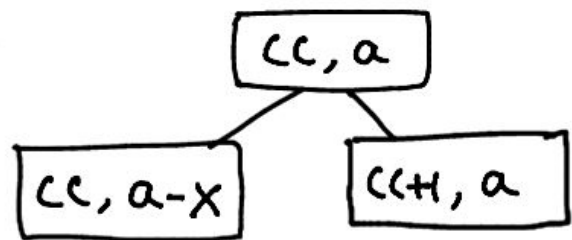
Given an array of coins & amount, find total number of ways / combinations to make up that amount.

Eg coins = [1, 2, 5]  
amount = 5

4 ways {  
1+1+1+1+1  
1+1+1+2  
1+2+2  
5

coins = [ \_  $\overset{x}{\curvearrowright}$  \_ \_ \_ ]  $x = \text{coins}[cc]$

current coin  $\curvearrowright$   $cc, a$   $\rightarrow$  amount



code →

```
1 class Solution {
2 public:
3
4     int totalWays(int currentIndex, vector<int>& coins, int amount, vector<vector<int>>&memo){
5         if(amount == 0) return 1; // amount==0 means that target is reached so return 1
6         if(currentIndex >= coins.size()) return 0; //if index is out of bounds then return 0
7
8         if(memo[currentIndex][amount] != -1) return memo[currentIndex][amount];
9
10        int consider = 0;
11        if(coins[currentIndex] <= amount){
12            consider = totalWays(currentIndex, coins, amount - coins[currentIndex], memo);
13        }
14        int notConsider = totalWays(currentIndex+1, coins, amount, memo);
15
16        memo[currentIndex][amount] = consider + notConsider;
17        return memo[currentIndex][amount];
18    }
19
20    int change(int amount, vector<int>& coins) {
21        vector<vector<int>> memo(coins.size()+1, vector<int>(amount+1, -1));
22        return totalWays(0, coins, amount, memo);
23    }
24 };
```

# 14 Coin Change →

(Similar to unbounded knapsack)

Given an array of coins & amount, find fewest number of coins to make up that amount, return -1 if its not possible

Eg coins = [1, 2, 5]  
amount = 11

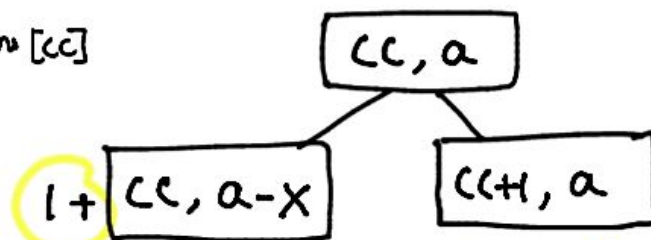
for 11 ⇒

$$\begin{array}{l} \overbrace{1+...+1}^{11 \text{ times}} = 11 \\ 1+2+2+2+2+2 = 11 \\ 1+5+5 = 11 \end{array}$$

out of all the ways last has min coins.

coins = [ -  $\overset{x}{\curvearrowright}$  - - - ]  $x = \text{coins}[cc]$

current coin  $\swarrow$   $cc, a$   $\searrow$  amount



↳ this contributes to counting coins.

code →

```

1 class Solution {
2 public:
3
4     int minimumCoins(int currentIndex, vector<int>& coins, int amount, vector<vector<int>>& memo){
5         if(amount == 0) return 0;
6         if(currentIndex >= coins.size()) return 100000; //Any Max Value outside boundary
7
8         if(memo[currentIndex][amount] != -1) return memo[currentIndex][amount];
9
10        int consider = 100000;
11        if(coins[currentIndex] <= amount){
12            consider = 1 + minimumCoins(currentIndex, coins, amount-coins[currentIndex], memo);
13        }
14
15        int notConsider = minimumCoins(currentIndex+1, coins, amount, memo);
16
17        memo[currentIndex][amount] = min(consider, notConsider);
18        return memo[currentIndex][amount];
19    }
20
21    int coinChange(vector<int>& coins, int amount) {
22
23        vector<vector<int>> memo(coins.size()+1, vector<int>(amount+1, -1));
24        int ans = minimumCoins(0, coins, amount, memo);
25
26        return (ans==100000)? -1 : ans;
27    }
28 };
  
```

### ⑮ Rod Cutting →

Given a rod of length  $N$  and array of prices. find the max value that can be obtained by cutting rod.

Eg  $N=8$  prices =  $[1, 5, 8, 9, 10, 17, 17, 20]$

1 2 3 4 5 6 7 8  
1 5 8 9 10 17 17 20  
0 1 2 3 4 5 6 7

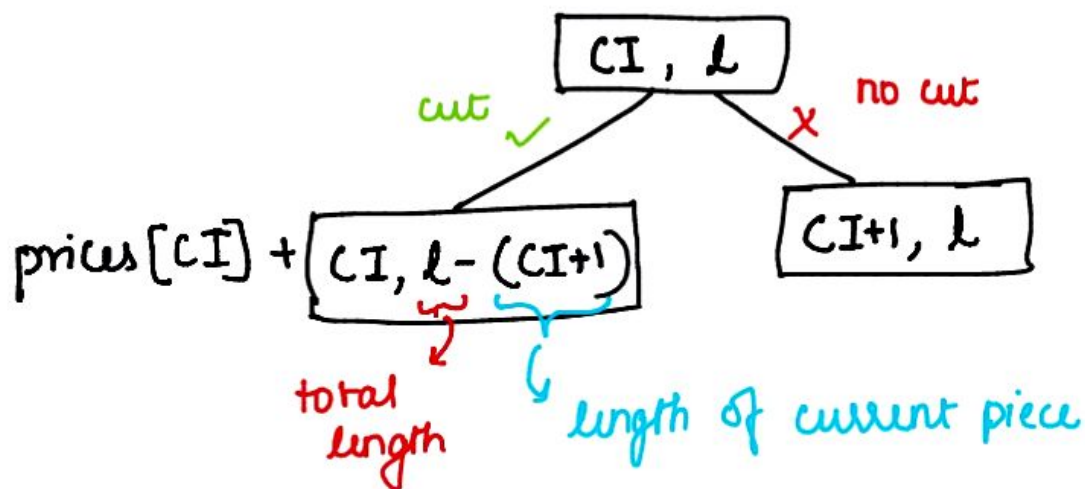
\* price of a piece is  $\text{prices}[CI]$ , whose length is  $CI+1$



if we cut our rod into 2 pieces of length 2, 6 we get max value of  $5+17$  i.e. 22.

→ there might be other ways, but this particular configuration returns max value.

\* At any instance length of current piece is  $CI+1$



code →

```
1
2 class Solution{
3     public:
4         int maxProfit(int price[],int currentIndex, int n, vector<vector<int>>&memo){
5             if(n==0) return 0;
6             if(currentIndex>=n) return 0;
7
8             if(memo[currentIndex][n]!=-1) return memo[currentIndex][n];
9
10            int selected = 0;
11            if(currentIndex+1<=n){
12                selected = price[currentIndex]+maxProfit(price, currentIndex, n-(currentIndex+1), memo);
13            }
14            int notSelected = maxProfit(price, currentIndex+1, n, memo);
15
16            memo[currentIndex][n] = max(selected, notSelected);
17            return memo[currentIndex][n];
18        }
19        int cutRod(int price[], int n) {
20            vector<vector<int>> memo(n+1, vector<int>(n+1,-1));
21            return maxProfit(price,0,n,memo);
22        }
23    };
```



# Dynamic Programming - 2

- Karun Karthik

## Contents

16. Best Time to Buy and Sell Stock
17. Best Time to Buy and Sell Stock II
18. Best Time to Buy and Sell Stock III
19. Best Time to Buy and Sell Stock IV
20. Best Time to Buy and Sell Stock with Cooldown
21. Best Time to Buy and Sell Stock with Transaction Fee
22. Jump Game
23. Jump Game II
24. Reach a given Score
25. Applications of Catalan Numbers
26. Nth Catalan Number
27. Number of Valid Parenthesis Expression
28. Unique Binary Search Trees

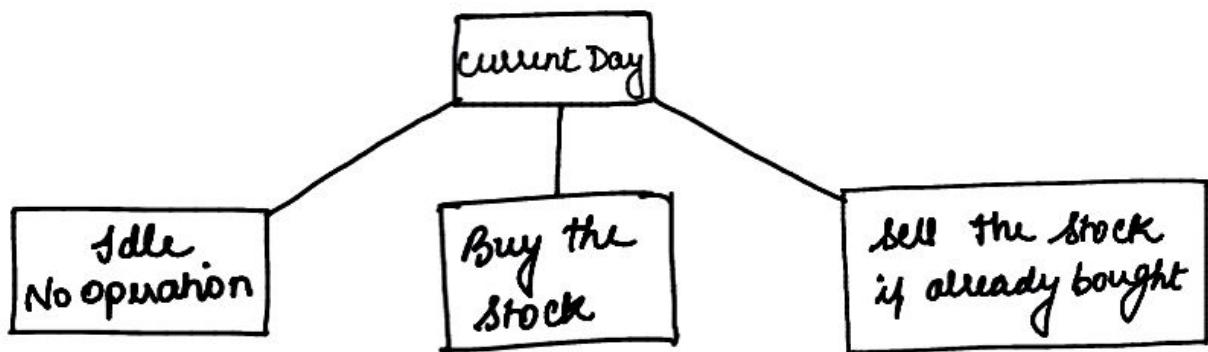
## ⑩ Best time to Buy & Sell Stock →

Given an array of prices, find the max profit if we are allowed to do one transaction

Eg

prices = [7, 1, 5, 3, 6, 4] → we get max profit when we buy at day 0 & sell on day 4  
                  0 1 2 3 4 5  
                  ⇒ profit = 6 - 1 = 5.

lets look at choices we have,



→ to handle the case that transaction could occur once, we use a variable called transaction = 1.

→ to handle these cases, we use a variable called canBuy.

→ once bought canBuy = false

→ once sold canBuy = true

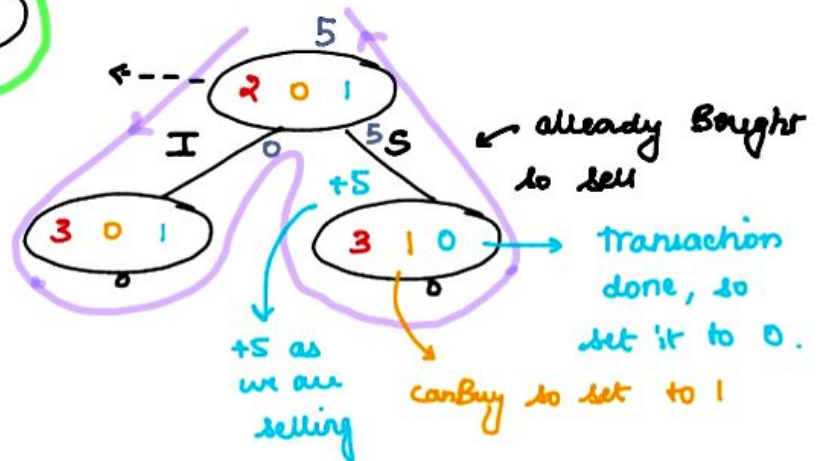
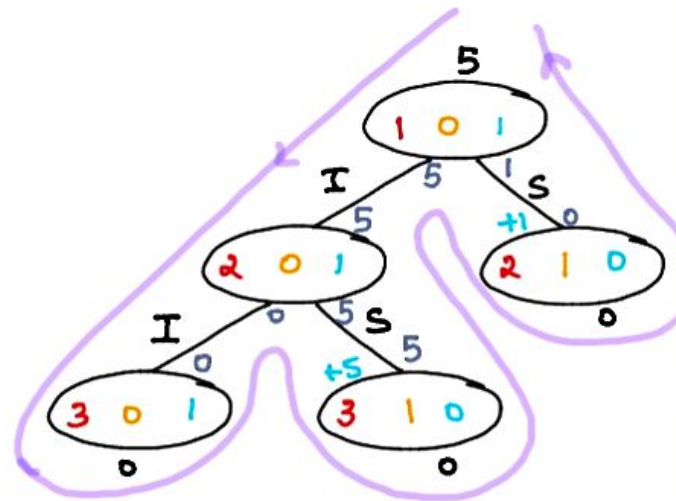
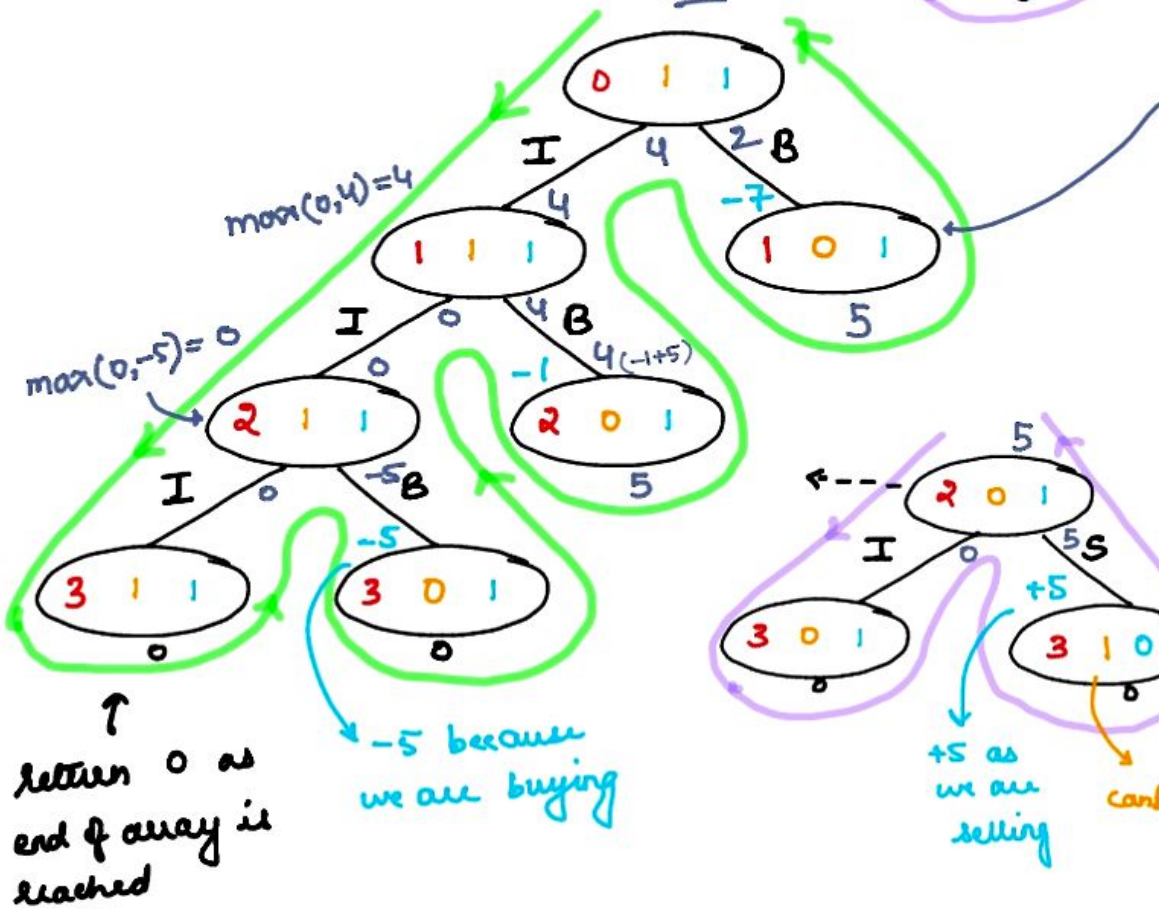
∴ Our recursive structure would be as follows →

current Day, canBuy, transaction

Eg  $[7, 1, 5]$   
0 1 2

Result →

4



code →

```
1  class Solution {
2  public:
3      int find(vector<int> &prices, int currDay, int k, bool canBuy, vector<vector<int>> &memo){
4
5          if(currDay >= prices.size() || k<=0 ) return 0;
6
7          if(memo[currDay][canBuy] != -1) return memo[currDay][canBuy];
8
9          if(canBuy)
10         {
11             int idle = find(prices, currDay+1, k, canBuy, memo);
12             int buy = -prices[currDay] + find(prices, currDay+1, k, !canBuy, memo);
13             return memo[currDay][canBuy] = max(buy, idle);
14         }
15         else
16         {
17             int idle = find(prices, currDay+1, k, canBuy, memo);
18             int sell = prices[currDay] + find(prices, currDay+1, k-1, !canBuy, memo);
19             return memo[currDay][canBuy] = max(sell, idle);
20         }
21     }
22     int maxProfit(vector<int>& prices) {
23         int n = prices.size();
24         vector<vector<int>> memo(n,vector<int> (2,-1));
25         // canBuy = true and transaction as k = 1
26         return find(prices,0,1,true,memo);
27     }
28 };
```



## ⑪ Best time to Buy & Sell Stock - II →

→ In this we can have many transactions that can be done.

Eg → prices = [7, 1, 5, 3, 6, 4]

↳ Buy on 1 & sell on 2      profit = 5 - 1 = 4

Buy on 3 & sell on 4      profit = 6 - 3 = 3

Total Profit = 7 Ans

code →

Remove the parameter K i.e transaction limit.

```
1 class Solution {
2 public:
3     int find(vector<int> &prices, int currDay, bool canBuy, vector<vector<int>> &memo){
4
5         if(currDay >= prices.size()) return 0;
6
7         if(memo[currDay][canBuy] != -1) return memo[currDay][canBuy];
8
9         if(canBuy)
10        {
11            int idle = find(prices, currDay+1, canBuy, memo);
12            int buy = -prices[currDay] + find(prices, currDay+1, !canBuy, memo);
13            return memo[currDay][canBuy] = max(buy, idle);
14        }
15        else
16        {
17            int idle = find(prices, currDay+1, canBuy, memo);
18            int sell = prices[currDay] + find(prices, currDay+1, !canBuy, memo);
19            return memo[currDay][canBuy] = max(sell, idle);
20        }
21    }
22    int maxProfit(vector<int>& prices) {
23        int n = prices.size();
24        vector<vector<int>> memo(n, vector<int> (2, -1));
25        // canBuy = true and transaction are infinite so ignore k
26        return find(prices, 0, true, memo);
27    }
28 };
```

### ⑬ Best time to Buy & Sell Stock - III →

In this maximum profit has to be achieved by making atmost 2 transactions.

Eg prices = [3, 3, 5, 0, 0, 3, 1, 4]

↳ Buy on 4 & sell on 5

$$\text{profit} = 3 - 0 = 3$$

Buy on 6 & sell on 7

$$\text{profit} = 4 - 1 = 3$$

$$\text{Total Profit} = \underline{6 \text{ Ans}}$$

code →

In the base condition is no. of transactions  $\geq 2$  then return 0.

(Line 6)

↳ ie possible transactions are when it is = 0, 1

```
1 class Solution {
2 public:
3     int find(vector<int> &prices, int currDay, int transaction, bool canBuy,
4             vector<vector<vector<int>>> &memo){
5
6         if(currDay >= prices.size() || transaction >= 2) return 0;
7
8         if(memo[currDay][canBuy][transaction] != -1) return memo[currDay][canBuy][transaction];
9
10        if(canBuy)
11        {
12            int idle = find(prices, currDay+1, transaction, canBuy, memo);
13            int buy = -prices[currDay] + find(prices, currDay+1, transaction, !canBuy, memo);
14            return memo[currDay][canBuy][transaction] = max(buy, idle);
15        }
16        else
17        {
18            int idle = find(prices, currDay+1, transaction, canBuy, memo);
19            int sell = prices[currDay] + find(prices, currDay+1, transaction+1, !canBuy, memo);
20            return memo[currDay][canBuy][transaction] = max(sell, idle);
21        }
22    }
23    int maxProfit(vector<int> &prices) {
24        int n = prices.size();
25        vector<vector<vector<int>>> memo(n, vector<vector<int>>>(2, vector<int>(2, -1)));
26        // canBuy = true and transactions are allowed 2 times
27        return find(prices, 0, 0, true, memo);
28    }
29 };
```

## ⑪ Best time to Buy & Sell Stock - IV →

This is a generalised version of previous problem, instead of limiting it to 2 transactions, we need to allow atmost  $k$  transactions.

code →

Pass  $k$  as an argument & use it to limit transaction in base condition. (Line 6)

```
1 class Solution {
2 public:
3     int find(vector<int> &prices, int currDay, int transaction, int k, bool canBuy,
4         vector<vector<vector<int>>> &memo){
5
6         if(currDay >= prices.size() || transaction>=k) return 0;
7
8         if(memo[currDay][canBuy][transaction] != -1) return memo[currDay][canBuy][transaction];
9
10        if(canBuy)
11        {
12            int idle = find(prices, currDay+1, transaction, k, canBuy, memo);
13            int buy = -prices[currDay] + find(prices, currDay+1, transaction, k, !canBuy, memo);
14            return memo[currDay][canBuy][transaction] = max(buy, idle);
15        }
16        else
17        {
18            int idle = find(prices, currDay+1, transaction, k, canBuy, memo);
19            int sell = prices[currDay] + find(prices, currDay+1, transaction+1, k, !canBuy, memo);
20            return memo[currDay][canBuy][transaction] = max(sell, idle);
21        }
22    }
23    int maxProfit(int k, vector<int>& prices) {
24        int n = prices.size();
25        vector<vector<vector<int>>> memo(n, vector<vector<int>>(2, vector<int>(k+1, -1)));
26        // canBuy = true and transactions are allowed atmost k times
27        return find(prices, 0, 0, k, true, memo);
28    }
29 };
```



## ②① Best time to Buy & Sell Stock with Cooldown →

In this, cooldown means that we cannot buy a stock on the immediate day after it is sold.

⇒ The day after sold should be skipped.

code →

To skip day after sell, increment the currDay by 2.  
(Line 18)

```
1  class Solution {
2  public:
3      int find(vector<int> &prices, int currDay, bool canBuy, vector<vector<int>> &memo){
4
5          if(currDay >= prices.size()) return 0;
6
7          if(memo[currDay][canBuy] != -1) return memo[currDay][canBuy];
8
9          if(canBuy)
10         {
11             int idle = find(prices, currDay+1, canBuy, memo);
12             int buy = -prices[currDay] + find(prices, currDay+1, !canBuy, memo);
13             return memo[currDay][canBuy] = max(buy, idle);
14         }
15         else
16         {
17             int idle = find(prices, currDay+1, canBuy, memo);
18             int sell = prices[currDay] + find(prices, currDay+2, !canBuy, memo);
19             return memo[currDay][canBuy] = max(sell, idle);
20         }
21     }
22     int maxProfit(vector<int> &prices) {
23         int n = prices.size();
24         vector<vector<int>> memo(n, vector<int> (2, -1));
25         // canBuy = true & transaction = infinite so ignore k & while sell, currDay +=2
26         return find(prices, 0, true, memo);
27     }
28 };
```



## ②1 Best time to Buy & Sell Stock with Transaction Fee →

In this variation, we don't have limit on transaction but while making a transaction i.e. selling it, some fee has to be paid i.e. transaction fee.

code →

Deduct the fee from the selling day's amount.

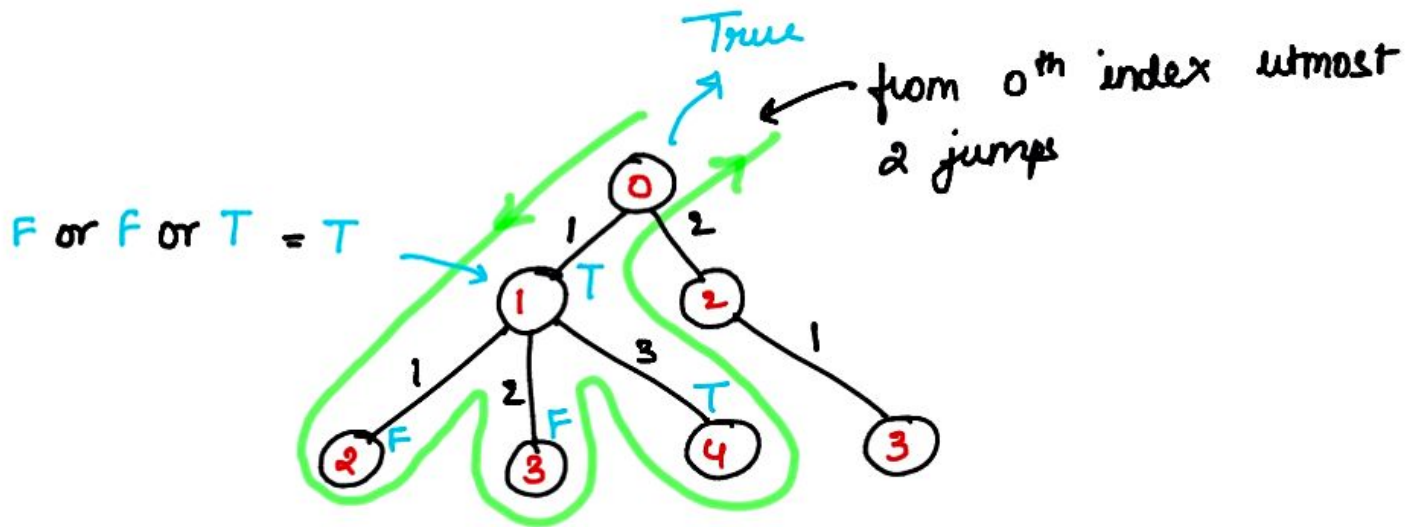
(Line 18)

```
1 class Solution {
2 public:
3     int find(vector<int> &prices, int currDay, int fee, bool canBuy, vector<vector<int>> &memo){
4
5         if(currDay >= prices.size()) return 0;
6
7         if(memo[currDay][canBuy] != -1) return memo[currDay][canBuy];
8
9         if(canBuy)
10        {
11            int idle = find(prices, currDay+1, fee, canBuy, memo);
12            int buy = -prices[currDay] + find(prices, currDay+1, fee, !canBuy, memo);
13            return memo[currDay][canBuy] = max(buy, idle);
14        }
15        else
16        {
17            int idle = find(prices, currDay+1, fee, canBuy, memo);
18            int sell = (prices[currDay]-fee) + find(prices, currDay+1, fee, !canBuy, memo);
19            return memo[currDay][canBuy] = max(sell, idle);
20        }
21    }
22
23    int maxProfit(vector<int>& prices, int fee) {
24        int n = prices.size();
25        vector<vector<int>> memo(n, vector<int> (2, -1));
26        // canBuy = true & transaction = infinite so ignore k & while selling deduce fee
27        return find(prices, 0, fee, true, memo);
28    }
29 };
```

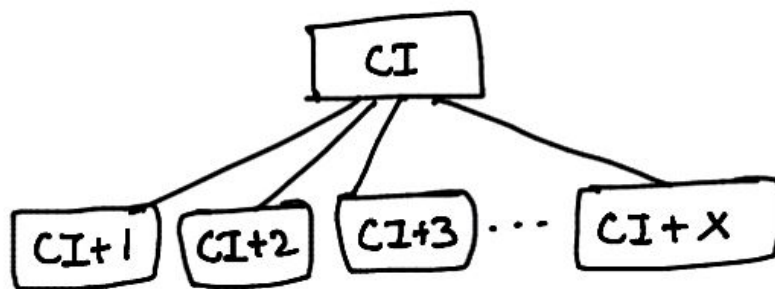
## ②② Jump Game →

Given array of nums which indicate max number of jump from any index. Return True if you can reach last index.

Eg nums = [2, 3, 1, 1, 4]  
0 1 2 3 4



Therefore,  $[- - \frac{x}{CI} - - -]$



} can be implemented using a for loop.

Note: Submitting DP solution gives TLE. This is just for understanding. Optimal solution involves Greedy approach.

$$T_c \rightarrow O(\underbrace{\max(\text{nums}[i])}_\text{max time for forloop} \times n)$$

code →

```

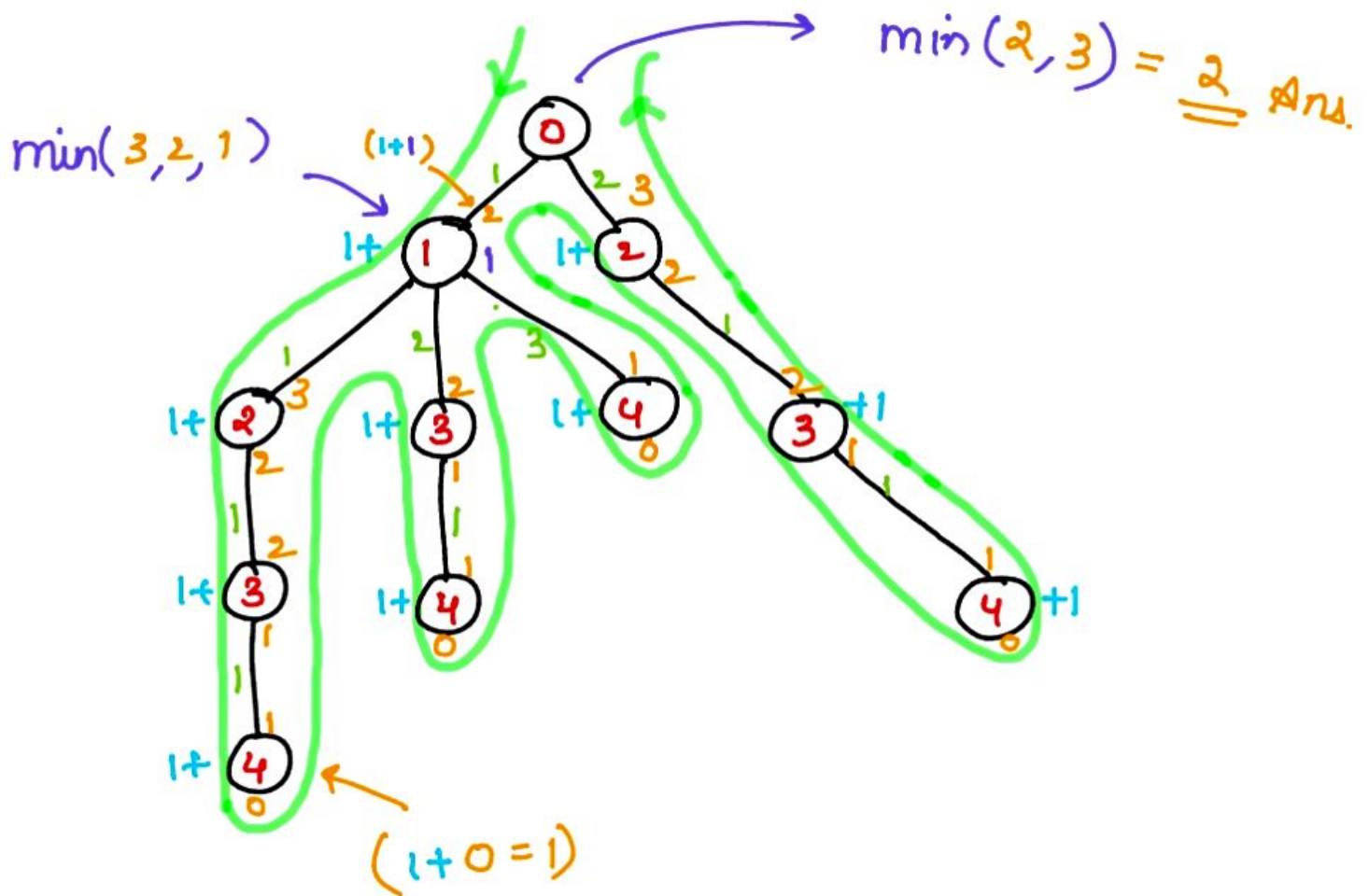
1  class Solution {
2  public:
3      bool isPossible(vector<int>&nums, int curr, unordered_map<int,bool>&memo)
4      {
5          if(curr >= nums.size()-1) return true;
6
7          int currKey = curr;
8
9          if(memo.find(currKey)!=memo.end()) return memo[currKey];
10
11         int currJump = nums[curr];
12
13         if(currJump >= nums.size() - curr) return true;
14
15         bool ans = false;
16
17         for(int i=1; i<=currJump; i++){
18             bool tempAns = isPossible(nums,curr+i,memo);
19             ans = ans || tempAns;
20         }
21         return memo[currKey] = ans;
22     }
23
24     bool canJump(vector<int>& nums){
25         unordered_map<int,bool>memo;
26         return isPossible(nums, 0, memo);
27     }
28 };

```

## ②③ Jump Game II →

Given array of nums which indicate max number of jump from any index. Reach last index in minimum number of moves.

Eg  $nums = [2, 3, 1, 1, 4]$   
0 1 2 3 4



→ If  $currentIndex \geq lastIndex$   
then return 0.

while returning add 1 for counting ways!



Code →

```
1  class Solution {
2  public:
3
4      int minJumps(vector<int>& nums,int curr,vector<int>&memo)
5      {
6          if( curr >= nums.size()-1) return 0;
7
8          int currKey = curr;
9          if(memo[currKey]!=-1) return memo[currKey];
10
11         int currJump = nums[curr];
12
13         // some large value
14         int ans = 10001;
15
16         for(int i=1;i<=currJump;i++){
17             int tempans = 1 + minJumps(nums,curr+i,memo);
18             ans = min(ans, tempans);
19         }
20         return memo[currKey] = ans;
21     }
22
23     int jump(vector<int>& nums) {
24         vector<int> memo(nums.size()+1,-1);
25         return minJumps(nums, 0, memo);
26     }
27 };
```

(24) Reach a given Score →

Given 3 scores  $[3, 5, 10]$  & 'n'.

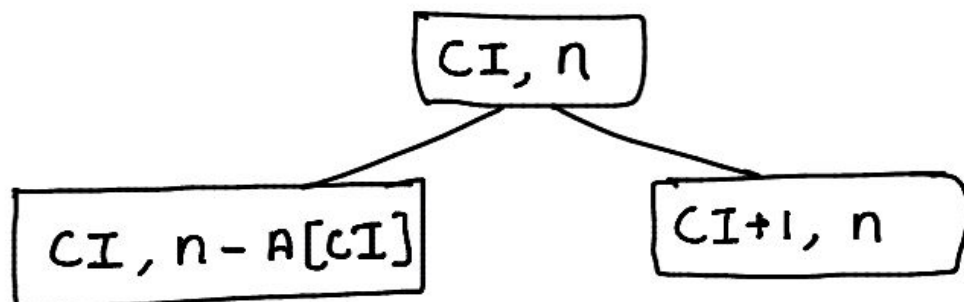
Return total number of ways to create n using the scores.

Ex  $n=8$  then no. of ways to create 8 from  $[3, 5, 10]$  is 1.  $(3+5)$

$n=13$  then no. of ways to create 13 from  $[3, 5, 10]$  is 2  $(3+5+5)$  &  $(3+10)$

$n=20$  then no. of ways to create 20 from  $[3, 5, 10]$  is 4  $(3+3+3+3+3+5)$  &  $(5+5+5+5)$   
&  $(5+5+10)$  &  $(10+10)$

∴ let say  $A = [3, 5, 10]$  then



code →

```
1  typedef long long LL;
2
3  LL ways(int curr, LL n, vector<int>&score, vector<vector<int>>&vec)
4  {
5      if(n==0) return 1;
6
7      if(curr>=score.size()) return 0;
8
9      if(vec[curr][n]!=-1) return vec[curr][n];
10
11     LL consider = 0;
12
13     if(score[curr]<=n)
14         consider = ways(curr,n-score[curr],score,vec);
15
16     LL notconsider = ways(curr+1,n,score,vec);
17
18     return vec[curr][n] = consider + notconsider;
19 }
20
21 LL count(LL n)
22 {
23     vector<int>score{3,5,10};
24     vector<vector<int>>vec(score.size(),vector<int>(1001,-1));
25     return ways(0,n,score,vec);
26 }
```

## ②⑤ Applications of Catalan Number →

Catalan Numbers are defined using the formula

$$C_n = \frac{(2n)!}{(n+1)! n!} = \prod_{k=2}^n \frac{n+k}{k} \text{ for } n \geq 0$$

this can be used recursively as follows,

$$C_{n+1} = \sum_{i=0}^n C_i C_{i-1} \quad \left. \vphantom{\sum_{i=0}^n} \right\} n \geq 0 \text{ \& } C_0 = 1$$

$$\rightarrow C_0 = \underline{\underline{1}}.$$

$$\rightarrow C_1 = \underline{\underline{1}}.$$

$$\rightarrow C_2 = C_0 \cdot C_1 + C_1 \cdot C_0 = 1 \cdot 1 + 1 \cdot 1 = \underline{\underline{2}}.$$

$$\rightarrow C_3 = C_0 C_2 + C_1 C_1 + C_2 C_0 = 1 \cdot 2 + 1 \cdot 1 + 2 \cdot 1 = \underline{\underline{5}}.$$

$$\begin{aligned} \rightarrow C_4 &= C_0 C_3 + C_1 C_2 + C_2 C_1 + C_3 C_0 \\ &= 1 \cdot 5 + 1 \cdot 2 + 2 \cdot 1 + 5 \cdot 1 = \underline{\underline{14}}. \end{aligned}$$

apps →

1. No. of possible BST with  $n$  keys.
2. No. of valid combinations for  $N$  pair of parentheses.



## ②6 N<sup>th</sup> Catalan Number →

To find N<sup>th</sup> Catalan Number we can use formula

$$C_{n+1} = \sum_{i=0}^n C_i C_{i-1} \quad \left. \vphantom{\sum_{i=0}^n} \right\} n \geq 0 \quad \& \quad C_0 = 1$$

↳ this can be implemented by

- i) having base condition for  $n == 0$  &  $n == 1$
- ii) using a loop to sum values from  $i = 0$  to  $n$

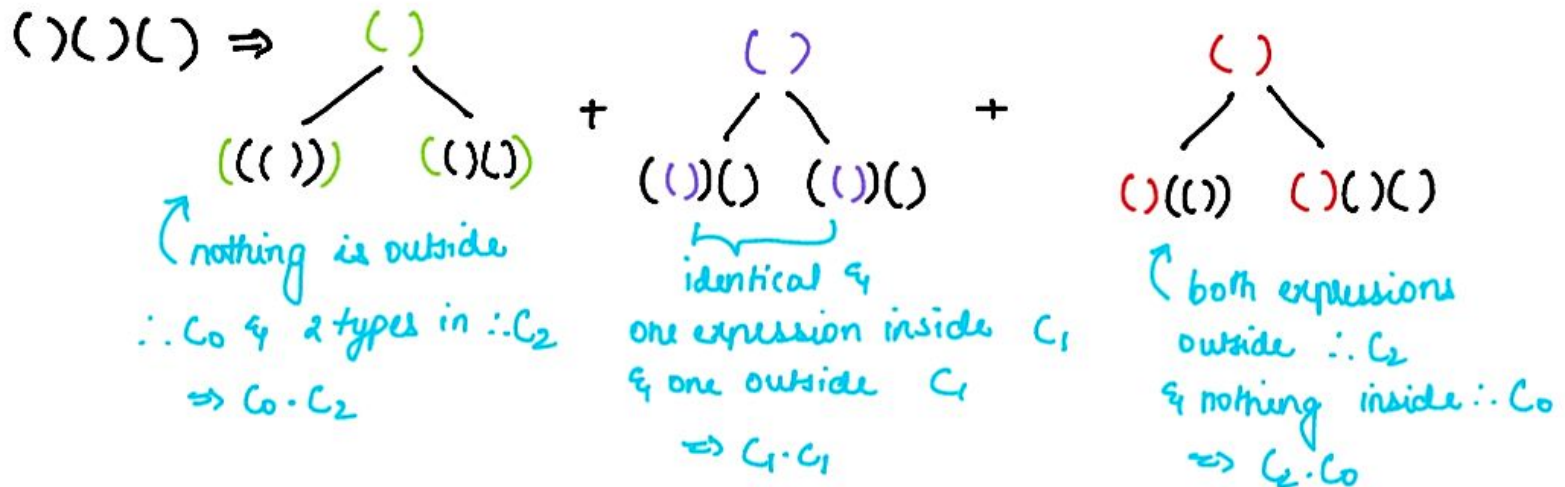
Code →

```
1 class Solution
2 {
3     public:
4     cpp_int ncatalan(int n, vector<cpp_int>& memo) {
5         if(n == 0 || n == 1) return 1;
6
7         int curr = n;
8         if(memo[curr] != -1) return memo[curr];
9
10        cpp_int catalan = 0;
11
12        for(int i=0; i<n; i++) {
13            catalan += ncatalan(i, memo)*ncatalan(n-i-1, memo);
14        }
15
16        memo[curr] = catalan;
17        return memo[curr];
18    }
19
20    cpp_int findCatalan(int n)
21    {
22        vector<cpp_int> memo(1001, -1);
23        return ncatalan(n, memo);
24    }
25 };
```

## (27) Number of valid Parenthesis Expression →

Given  $N$ , find total number of ways in which we can arrange  $N$  pair of parenthesis in a balanced way.

Eg  $N=4 \Rightarrow ()()(), ()(()) , ((())) , ((() )) \therefore \text{res} = 4$



$\Rightarrow C_0 \cdot C_2 + C_1 \cdot C_1 + C_2 \cdot C_0 = C_3 \Rightarrow \text{for } n \text{ we need to find } \text{ncatalan}(n/2)$

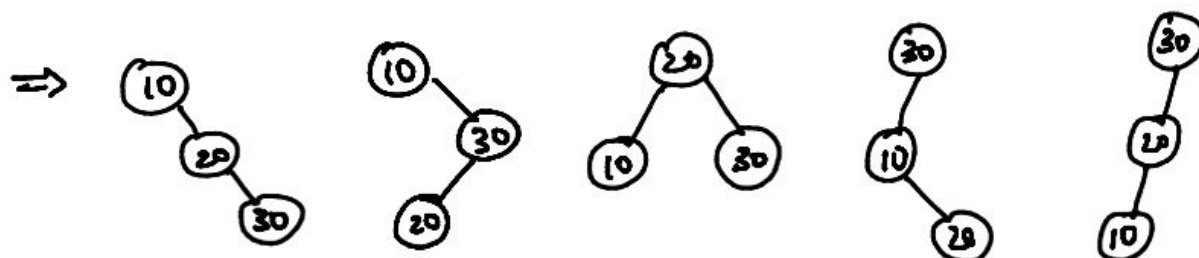
Code →

```
1 #include<bits/stdc++.h>
2 using namespace std;
3
4 int ncatalan(int n, unordered_map<int,int>& memo) {
5     if(n == 0 || n == 1) return 1;
6
7     int curr = n;
8     if(memo[curr] != -1) return memo[curr];
9
10    int catalan = 0;
11
12    for(int i=0; i<n; i++) {
13        catalan += ncatalan(i, memo)*ncatalan(n-i-1, memo);
14    }
15
16    memo[curr] = catalan;
17    return memo[curr];
18 }
19
20 int countValidParenthesis(int n)
21 {
22     unordered_map<int,int> memo;
23     return ncatalan(n/2, memo);
24 }
25
26 int main(){
27     int n;
28     cin>>n;
29     cout<<countValidParenthesis(n);
30 }
```

## ②8 Unique Binary Search Trees →

given integer  $N$ , return no. of unique BST that can be formed.

Eg  $n=3$  & let's say elements are  $[10, 20, 30]$



∴ For  $n=3$ , the result is 5.

∴ The catalan number gives us the result.

code →

```
1 class Solution {
2 public:
3
4     int uniqueBST(int n, vector<int>& memo)
5     {
6         if(n==0||n==1) return 1;
7
8         if( memo[n]!=-1)return memo[n];
9
10        int ans = 0;
11        for(int i=0;i<n;i++)
12            ans += uniqueBST(i,memo)*uniqueBST(n-i-1,memo);
13
14        return memo[n] = ans;
15    }
16
17    int numTrees(int n) {
18        vector<int> memo(n+1,-1);
19        return uniqueBST(n, memo);
20    }
21 };
```